JJEE

Volume 2, Number 3, 2016
Pages 215-230

Jordan Journal of Electrical Engineering

ISSN (Print): 2409-9600, ISSN (Online): 2409-9619

Optimized Parallel Architecture of Kalman Filter for Radar Tracking Applications

Amin A. Jarrah

Department of Computer Engineering, Yarmouk University, Irbid, Jordan
e-mail: amin.jarrah@yu.edu.jo

Received: April 14, 2016 *Accepted: May 7, 2016*

Abstract— Kalman filter has been proven to be a very effective method to identify targets in an efficient and accurate manner. It provides efficient estimations when the precise nature of the modeled system is unknown in the presence of measurement and process noise. However, Kalman filter is computationally extensive especially in Multi Target Tracking (MTT) radar system. Therefore, it is desirable to apply it on advanced parallel architecture such as FPGA, GPU, and multi-cores to increase performance and achieve real time requirements. In this paper, we present an efficient parallel architecture of Kalman filter on different platforms such as FPGA, GPU, and multi-core. Kalman filter operations are carried out on a single core CPU before they are decomposed, parallelized, scheduled, and mapped into FPGA and GPU platforms. Different optimization techniques for both the computation and memory utilization are adopted and applied to achieve high performance. The experimental results show the viability of using FPGA and GPU platforms to perform signal processing in real time. Parallel architectures can significantly outperform an equivalent sequential implementation due to their pipelined architecture, custom functionality of VLSI ASIC devices, flexibility, and adaptability. Our simulation results indicate that the achieved speed-up of FPGA and GPU over the sequential one is improved by up to 37.76 and 31.93, respectively.

Keywords— Field-programmable gate array (FPGA), Graphic processing unit (GPU), Kalman filter, Optimization techniques, Parallel architecture.

I. INTRODUCTION

Tracking provides the current and estimated flight path of any target of interest. Tracking targets using radars may be very challenging due to the randomness of the data and presence of large amount of noise commonly known as clutter [1]-[3]. Tracking can also be considered as a filtering operation for removing unwanted targets. Due to the randomness in the radar data, tracking algorithms play a vital role in target detection and clutter removal [2]. Therefore, it is desirable to implement the Kalman filter which is a successive process for "predict - rectify" [4]. The filter does not need to save large data during the solving process; and the calculations of the new values are performed as soon as new data is observed.

Kalman filter [5]-[8] is an efficient technique to estimate the state of a system from a noisy environment. It involves two main steps: the prediction step in which the state of the system is predicted; and the measurement step in which the estimation of the system state from noisy measurement is refined and corrected. There are many variants of Kalman filter [5] that are widely used for applications relying on estimation. In this work, Kalman filter has been used to track a desired object for radar application. This will help predict the object's future location and associate multiple objects with their corresponding tracks. Fig. 1 shows an example of Kalman filter processes to estimate the position from a radar application.

However, Kalman filter is a complex and precise algorithm to approach the radar tracking problem [9]. Radar systems perform high complex computations and require high speed architecture to meet the real time constraints especially in MTT systems and track many targets simultaneously. So, it is desirable to implement Kalman filter on high speed parallel architecture such as Field Programmable Gate Array (FPGA), Graphic Processing Unit (GPU), and multi-core to achieve the real time requirements. In this work, Kalman filter is decomposed, parallelized, scheduled, and mapped into FPGA and GPU platforms where different optimization techniques for both the computation and memory utilization are adopted and applied to achieve high performance.

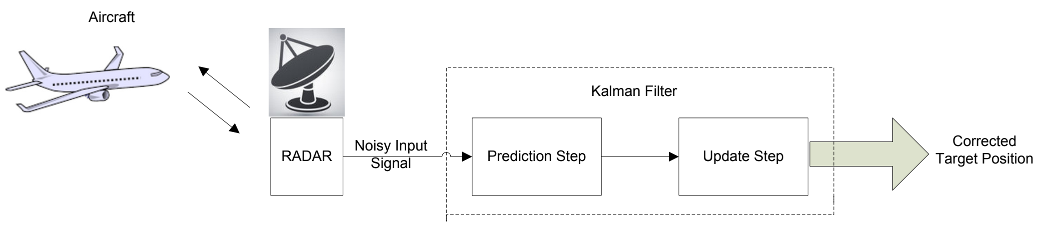


Fig. 1. Kalman filter processes to estimate the position from a radar application

The implementation of the Kalman filter fall into two major categories: software implementation using parallel processing and dedicated hardware implementations using customized Very Large-Scale Integration (VLSI) devices. Each implementation category shows different trade-offs in terms of latency, area, power consumption, cost, and flexibility. There are two main drawbacks for software based implementation:

- Firstly, programming multi-cores system is difficult and time consuming.
- Secondly, it is not cost effective since parallelization needs more processing elements.

On the other hand, hardware implementation has the property of its custom functionality of Application Specific Integrated Circuit (ASIC) devices, flexibility, adaptability, low cost, and pipelined architecture. The design and implementation using FPGA parallel platform allows for a good reutilization of expensive hardware resources. Moreover, the implementation using FPGAs not only includes a larger hardware area, but also embedded processors and memory resources. This option offers versatility in running diverse software applications on embedded processors, while taking advantage of reconfigurable hardware resources, all on the same chip package. Optimized parallel architectures using both parallel processing and hardware implementation have been performed to achieve real time requirements and show the best architecture for Kalman filter.

Kalman filter has different versions and configurations based on the applied application and objective. Most of the existing techniques for parallelizing Kalman filter focus on parallelizing matrix operations [10], [11]. However, the techniques in both [10] and [11] do not achieve high performance when the matrix dimensions are not large to fill the GPU pipelines. In [12], a new method was examined for parallelizing Kalman filter, where the data dependency was broken through re-organizing calculations. However, implementation is for a specific kind of Kalman applications. In [13], a new efficient implementation for parallelizing matrix operations has been examined for parallelizing Kalman filter. Other methods [14], [15] use parallel reduction techniques to get great speed-up. However, our contribution in this paper will include:

- Optimized parallel architecture of Kalman filter on GPU has been performed.
- Optimized parallel architecture of Kalman filter on FPGA has been performed.
- Different optimized techniques such as pipelining and dataflow techniques are applied by allowing the concurrent execution of operations to improve throughput and latency. This implementation achieved high speed with an awareness of power, area and cost.
- Different optimized techniques for memory utilization have been adopted and applied to achieve high performance.

The remainder of this paper is organized as follows: Section 2 details the Kalman filter theory and its characteristics; optimization techniques for GPU platform are discussed in Section 3; Section 4 provides different optimization techniques for FPGA platform; simulations and synthesis results are presented in Section 5; and Section 6 provides a conclusion of the study.

II. KALMAN FILTER OPERATION

Kalman filter provides stochastic estimation in a noisy environment [5]. It operates one estimating state by using recursive time and measurement updates over time. Noise effect on the system decreases due to recursive cycles which finally lead to the true value of the measurement [16]. The application of filters is to extract information from a given signal. Signals are represented in [5] as:

$$y_k = a_k x_k + n_k \tag{1}$$

where y_k is the observed signal; a_k is the gain; x_k is the information signal; and n_k is the noise. Thus, the purpose of the filter is to estimate x_k.

The error is defined as the difference between the estimated value $\tilde{x}_k$ and x_k. The process of a system [5] is assumed to be:

$$x_{k+1} = Ax_k + w_k \tag{2}$$

where x_k is state vector at time *k*; A is state transition matrix of process from time *k* to *k*+1; and w_k is the process noise. Observation on x_k [5] can be given as:

$$z_k = Hx_k + v_k \tag{3}$$

where z_k is the actual measurement; H is a transformation vector between the state and the measurement; and v_k is the measurement noise. It is assumed that the process and measurement noises are white uncorrelated noises. The filter should be designed to minimize the mean square error. The two noises are assumed to be stationary; and the covariance [5] is given as:

$$Q = E[w_k w_k^T] \tag{4}$$

$$R = E[v_k v_k^T] \tag{5}$$

Mean square error [5] is:

$$P_k = E[e_k e_k^T] \tag{6}$$

P_k is the error covariance [5] given as:

$$P_k = E[e_k e_k^T] = E[(x_k - \tilde{x}_k)(x_k - \tilde{x}_k)^T] \tag{7}$$

where $\tilde{x}_k$ is the estimate of x_k. Let the prior estimate of $\tilde{x}_k$ be as $\tilde{x}_k^{'}$. An equation combining measurements with previous estimated data is given as:

$$x_k = \tilde{x}_k^{'} + K_k(z_k - H\tilde{x}_k^{'}) \tag{8}$$

where K_k is the Kalman gain; $z_k - H\tilde{x}_k^{'}$ is the innovation or measurement module. Substituting (3) in (8), we obtain:

$$x_k = \tilde{x}_k^{'} + K_k(Hx_k + v_k - H\tilde{x}_k^{'}) \tag{9}$$

By substituting (9) in (7):

$$P_k = E[[(I - K_kH)(x_k - \tilde{x}_k^{'}) - K_kv_k][(I - K_kH)(x_k - \tilde{x}_k^{'}) - K_kv_k]]^T \tag{10}$$

where $x_k - \tilde{x}_k^{'}$ is the error of prior estimates that is uncorrelated with measurement noise and (10) is re-written as:

$$P_k = (I - K_kH)E[(x_k - \tilde{x}_k^{'})(x_k - \tilde{x}_k^{'})^T](I - K_kH) + K_kE[v_kv_k^T]K_k^T \tag{11}$$

By substituting (5) and (7) in (11):

$$P_k = (I - K_kH)P_k^{'}(I - K_kH)^T + K_kRK_k^T \tag{12}$$

where $P_k^{'}$ is prior estimate of P_k.
State projection [5] is calculated from:

$$\tilde{x}_{k+1}^{'} = A\tilde{x}_k \tag{13}$$

It is important in a recursion process to transfer the error covariance matrix to the next time interval [5] given as:

$$e_{k+1}^{'} = x_{k+1} - \tilde{x}_{k+1}^{'} = (Ax_k + w_k) - A\tilde{x}_k = Ae_k + w_k \tag{14}$$

Hence;

$$P_{k+1}^{'} = AP_kA^T + Q \tag{15}$$

The Kalman filter equations include the prediction and update equations [5]. The prediction of the estimate is given as:

$$\tilde{x}_k^{'} = A\tilde{x}_{k-1} \tag{16}$$

$$P_{k+1}^{'} = AP_kA^T + Q \tag{17}$$

where $\tilde{x}_k^{'}$ shows the predicted position of a target in the next state; A is the state transformation matrix; $\tilde{x}_k$ is the observed position of the target in the current state; $P_k^{'}$ is the predicted covariance; P_k is the covariance of the target position; and Q is the process covariance noise.
The updated equations of Kalman filter [5] are given by:

$$K_k = P_k^{'}H^T(HP_k^{'}H^T + R)^{-1} \tag{18}$$

$$x_k = \tilde{x}_k^{'} + K_k(z_k - H\tilde{x}_k^{'}) \tag{19}$$

$$P_k = (I - K_k H)P_k^{'}(I - K_k H)^T + K_k R K_k^T \tag{20}$$

where K_k is the Kalman gain; H is the transformation matrix; R is the actual measurement; z_k is the update position of the target; and P_k is the updated covariance matrix.
The prediction stage tracks the position of the target in the next state, whereas the measurement step updates the position based on z_k, the position of the target in the current state. The gain and error estimation covariance becomes stable with time if the noise level is assumed to be constant. Noise assumptions play a very important role in the filter design [6]. The Kalman filter was considered to give the best estimation for linear data. Its popularity is ascribed to its optimality, easy implementation. It yields good results theoretically and practically. The structure in Fig. 2 shows the architecture of the Kalman filter.

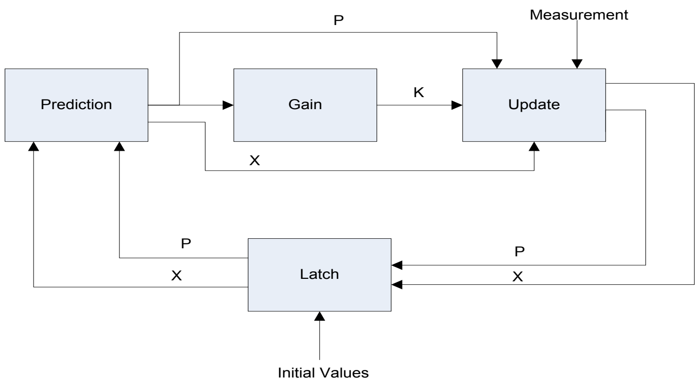


Fig. 2. Kalman filter design

The association of parameters within the prediction and update stages is shown in Fig. 3.

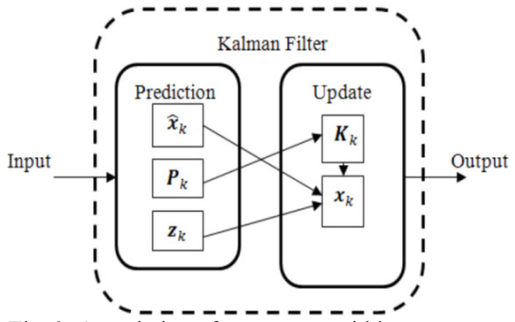


Fig. 3. Association of parameters within two stages

Kalman filter in our model estimates values, performs operations for both X and Y positions and tracks both directions toward the radar and far away from the radar. In this study the four blocks operations (X for both directions and Y for both directions) for a large number of targets are completely parallelized and mapped to different cores. The nearest neighbor data association between frames is performed in our model to match the estimated values with the targets in the current frame as shown in Fig. 4.

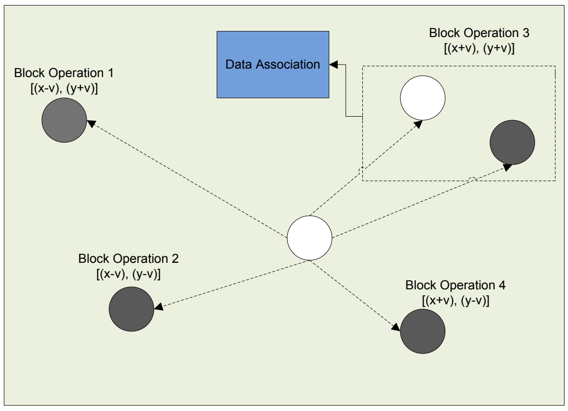


Fig. 4. Four blocks operations of Kalman filter design (X for both directions and Y for both directions)

III. GPU Architecture and Optimization Techniques

A. NVIDIA GForce GTX 260 GPU Architecture

Graphic Processing Unit (GPU) combines high bandwidth memories and hardware that perform floating point arithmetic at significantly higher rates than conventional CPUs. This makes the GPU to be very attractive for highly parallel computation algorithms.

NVIDIA GForce GTX 260 [17] GPU architecture is used in our work. GTX 260 GPU architecture contains hundreds of cores, and each core contains threads. Our target is to efficiently decompose the Kalman steps into tasks; each task can be executed on a different core; and each task contains threads which are executed simultaneously.

Each core of GTX 260 GPU architecture contains eight stream processors. The core runs in Single Instruction Multiple Data (SIMD) manner where all stream processors of a core execute the same instruction but operate on different data. There are threads, thread blocks, and grids of thread blocks that differentiate themselves based on memory access and kernel execution. A thread block is a group of threads that is able to cooperate with each other and communicate via the per-Block shared memory. Each block supports as many as 512 concurrent threads, each of which has a separate access to individual memory, counters, registers, etc. Each grid is considered as an array of thread blocks that are running the same kernel, and they are able to read and write from a global memory.

Computation on the GPU proceeds as follows:

- The user allocates memory on the GPU
- The user copies the data to the GPU
- The user specifies a program to execute on the GPU's cores
- And after execution, the user copies the data back to the CPU.

B. GPU Parallelization Techniques

There are three main steps in Kalman filter: prediction step, measurement step and update step. Each step needs high computation since the prediction, measurement or update process must be performed for a large number of targets. Moreover, Kalman filter in our design performs these operations for both X and Y positions and tracks objects coming toward the radar or far away from the radar. The four blocks operations (X for both directions and Y for both directions) for a large number of targets should be decomposed, scheduled, parallelized, and mapped efficiently into a parallel platform to achieve high performance. The four block operations are implemented as four functions (for both X and Y positions and for both directions) on different cores to be executed in parallel due to the lack of dependency between them. This makes each function to be executed as fast as possible to reduce the overall latency. However, we cannot execute all the Kalman filter steps for a certain function in parallel due to the dependency between its operations. The measurement steps cannot be executed until the prediction statement is performed; and the update step cannot also be executed until the measurement step is performed since there is a true data dependency Read After Write (RAW) between them. Without applying some parallelization techniques, prediction statement must serially execute and complete all its operations before measurement statement can begin reducing the overall throughput and increasing the latency. Therefore, different parallelization techniques are adopted and applied in the following sub-sections between the internal operations and steps for each block of Kalman filter. This will help execute all the operations for a certain function in parallel as much as possible by spreading each computation on a thread on a specific core. This contributes to improving both the throughput and the latency. There are three main operations in Kalman filter that need to be performed efficiently:

B.1. Matrix and vector transposition

Kalman filter requires performing transpose of some vectors and matrices through its computation. This requires some computation time and additional hardware resources to store the result. Loop blocking technique [18] is also adopted and applied in our model. It is a loop transformation which increases the depth of a loop nest by adding additional loops to the loop nest as shown in Fig. 5. Loop blocking optimization technique is primarily used to improve data locality by enhancing the reuse of the data in cache [18].

Original Code	*Loop Blocking Code*
1. $for\ i = 1\ to\ m$	1. $for\ i = 1\ to\ m\ by\ B$
2. $for\ j = 1\ to\ m$	2. $for\ j = 1\ to\ m\ by\ B$
3. $Mat_A[i,j] = Mat_B$[j,i]	3. $for\ k = i\ to\ \min(i + B - 1, m)$
4. $end\ for$	4. $for\ m = j\ to\ \min(j + B - 1, m)$
5. $end\ for$	5. $Mat_A\ [k,m] = Mat_B[m,k]$
	6. $end\ for$
	7. $end\ for$
	8. $end\ for$
	9. $end\ for$

Fig. 5. Loop blocking for matrix transposition

However, transpose operation changes the locations of the values. We can only modify the indices of the code loops without performing the transpose operation. This will improve performance in terms of computation time, hardware storage, and power dissipation as shown in the following code:

```
Function (X^H Y)
Declare ← X [] [], Y[][], Result [] [], Sum;
For k ←0 to M {
For i ←0 to M {
For j← 0 to N
            {Sum = Sum + X [j] [k] * Y [j] [i];}
    Result [k] [i] = Sum;
    Sum = 0;}}
```

B.2. Matrix to matrix multiplication which can be parallelized in two ways:

- Blocking technique

This technique is performed by decomposing the matrices into blocks. Each core maintains a block of A matrix and a block of B matrix as shown in Fig. 6. However, this requires synchronization mechanisms to combine the result back which is expensive operation.

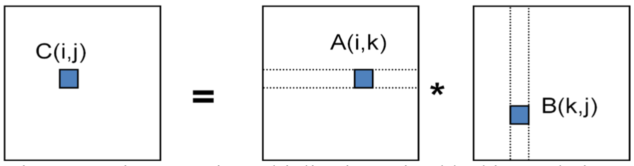


Fig. 6. Matrix to matrix multiplication using blocking technique

- Row-major techniques

This technique is performed by assigning one row and one column multiplication process for each core. Each core only multiplies one row by one column as shown in Fig. 7. This will remove the synchronization mechanism in the blocking technique. This technique is adopted in our work because the required matrices dimension of Kalman filter is not large enough to fill all the GPU pipelines.

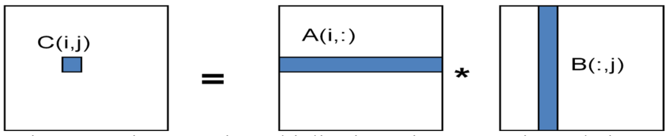


Fig. 7. Matrix to matrix multiplication using row-major technique

- Iterative map reduce matrix multiplication

The numerous matrix multiplications can also be implemented and parallized using Iterative Map Reduce algorithm [19]. In the iterative map reduce approach, each iteration produces an element of the resultant matrix obtained by manipulating the rows of input matrix and a specific column of the weight matrix. Fig. 8 shows that the iterative-based implementation requires three nested loops to produce the resultant matrix.

Concerning iterative MR, the nested loop in map reduce technique can be better executed by either unrolling or partially unrolling different iterations. The unrolling factor essentially depends on the resource mapping of the involved matrices and concurrency capability of read from the mapped resource.

```
1. INITIALIZATION:
   Input matrices: X[n×m] and W[m×p]
   Let y be the resultant matrix of the size
(n×p)
2. PARALLELIZATION:
   For i from 1 to n:
      For j from 1 to p:
         Let sum = 0
         For k from 1 to m:
            Set sum ← sum + $X_{ik}$ × $W_{kj}$
         Set $y_{ij}$ ← sum
```
Fig. 8. Pseudo code of iterative map reduce matrix multiplication

B.3. Matrix to vector multiplication

This operation is performed by applying row-major technique as in matrix to matrix multiplication as shown in Fig. 9.

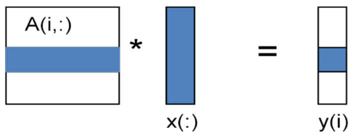


Fig. 9. Matrix vector multiplication process using row-major technique

C. Memory Optimization Techniques

C.1. Pre-fetching technique

Based on the pseudo code and memory accesses of the matrix vector multiplication in Fig. 10 and 11, the number of slow memory references is $2m+m^2$; and the number of arithmetic operations is m^2.

```
Implementation of y=A*x
1.  for $i = 1$ to $m$
2.    for $j = 1$ to $m$
3.        $y(i) = A(i,j) * x(j)$
4.    end for
5.    end for
```

Fig. 10. Matrix vector multiplication pseudo code

```
{read $x(1:m)$ into register}
for $i = 1:m$
{read row $i$ of $A$ into register}
for $j = 1:m$
$y(i) = A(i,j) * x(j)$
{write $y(1:m)$ back to slow memory}
```

Fig. 11. Memory accesses of matrix vector multiplication process

Accordingly, we need to optimize the code in respect to the arithmetic operations and memory references as matrix to matrix/vector multiplication will be limited by the slow memory speed. We applied the pre-fetching technique to reduce the demands on memory accesses. This is performed by exploiting multiple registers to access the element in the slow memory accesses before requiring it as shown in Fig. 12.

```
For (…) {
  Y [ ] = A[0,0]*x[0] + A[1,0]*x[1] + A[2,0]*x[2] +
…etc;
}
      ⬇
float a1 A[0,0];
float a2 A[1,0];
float a3 A[2,0];

For ( … ) {
  Y[ ] = a1*x[0]+ a2*x[1] + a3*x[2];
}
```

Fig. 12. Arithmetic operations and memory references using pre-fetching technique

C.2. Copy optimization technique

The copy optimization technique is adopted and applied to reduce the cache conflicts and improve the spatial locality as shown in Fig. 13. This method copies the non contiguous data into a contiguous area of memory. Each word of data block will be mapped to its own cache location to avoid cache conflicts within the data block.

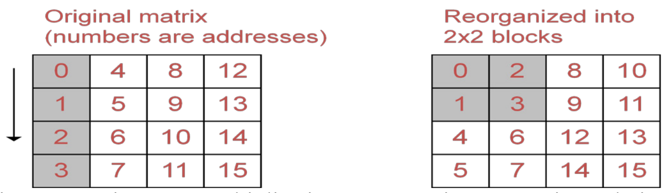


Fig. 13. Matrix vector multiplication process using row-major technique

D. *Coalescing Global Memory Access Technique*

Global memory resides in the GPU device. Its memory is larger than other memories of the GPU; and it is accessible by all the threads of the GPU. However, it has high latency. The operations on the GPU device are issued per wrap. It is very important to make the wrap access the memory in a contiguous region to improve bus utilization instead of scattered address patterns with large strides between threads as shown in Fig. 14. Coalescing global memory access technique [20] within a wrap is applied in our work to improve performance as shown in Fig. 15.

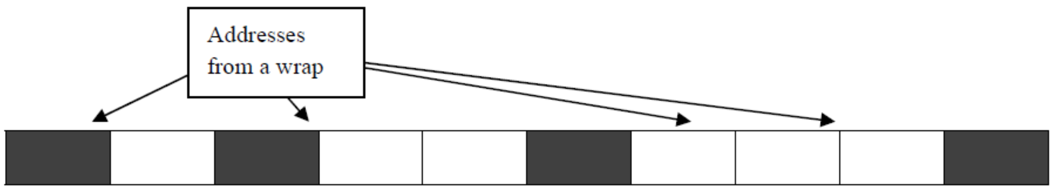


Fig. 14. Scattered address patterns between threads

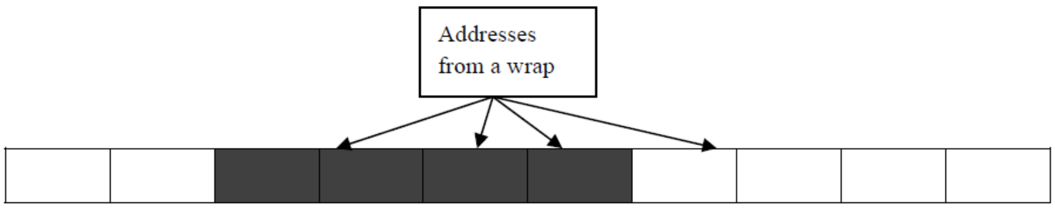


Fig. 15. Consecutive address patterns by applying coalescing global memory access technique

E. *Exploit GPU Fast Memory*

GPU memory hierarchy consists mainly of five levels: register, cache L1, cache L2, shared memory, and global memory. It is very important to use and exploit the GPU memory hierarchy to get how fast each memory is.

Register is the fastest memory on the GPU where each thread has its own storage. Shared memory and cache L1 have low latency and high bandwidth. Cache L2 is slower than the memories mentioned above; and the global memory is large. All threads have access to which has higher latency. Shared memory is very fast since it is resided on GPU card, but it has a small size. However, it is very effective in our model since the matrices dimensions of Kalman filter are not large. We tried to exploit some data of the filter into the shared memory; and we have achieved 14% better performance.

F. Minimize Communication between CPU and GPU

In order to parallelize any task on GPU efficiently, we need to identify and parallelize the hotspots on serial CPU code. Communication between the host (CPU) and the device (GPU) is expensive. All the tasks in our work are processed by GPU. CPU needs to communicate with GPU after the work has been processed.

IV. FPGA OPTIMIZED TECHNIQUES

Fig. 16 shows the hardware architecture to implement the Kalman filter along with the I/O communications. For data input, we use a Double Data Rate (DDR2) Synchronous Dynamic Random Access Memory (SDRAM) and a Direct Memory Access (DMA) which is used to access the external memory and store data in BRAM during the first stage. In the second stage, the computation process of Kalman filter which consists mainly of three steps begins the prediction of the target position, computation of the covariance and Kalman gain, and the update prediction of the target. Finally, the updated value is returned and stored in an external memory.

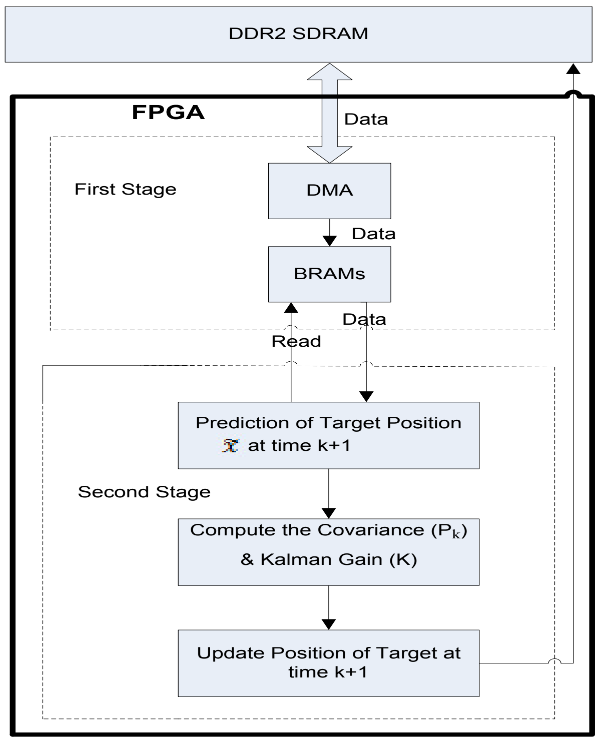

DDR2 SDRAM
FPGA
Data
First Stage
DMA
Data
BRAMs
Data
Read
Prediction of Target Position $\hat{x}$ at time k+1
Second Stage
Compute the Covariance (P_k) & Kalman Gain (K)
Update Position of Target at time k+1

Fig. 16. Hardware architecture adopted to implement the Kalman filter.

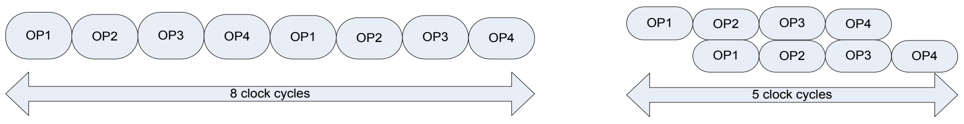
The same parallelization techniques on GPU are applied on the FPGA architecture. Each step of Kalman filter for both X and Y positions and for both directions are executed in parallel by applying both unrolling and pipelining techniques. The results of both techniques with different features are shown in Table 1.

Table 1 shows that the loop unrolling technique achieves the best performance in terms of latency. However, it uses a huge number of flip flops, lookup tables, and high power dissipation. In comparison, the performance of the pipelining technique is approximately similar to the loop unrolling. We have chosen pipelining technique since it achieves a better performance without degrading other performance parameters such as memory and power dissipation.

TABLE 1
COMPARISON OF APPLYING DIFFERENT OPTIMIZATION TECHNIQUES OF THE PREDICTION STEP FOR BOTH X AND Y POSITIONS ON ARTIX7 (XA7A100T CSG324) -1Q

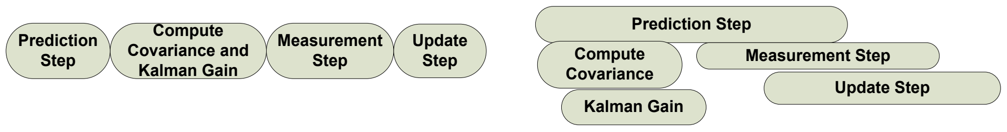
	Without applying any optimization	Applying Loop pipelining	Applying Loop Unrolling
Latency (cycles)	21005	1120	1090
Clock period (ns)	7.21	8.2	8.2
# of Flip Flop	620	4528	135008
# of Lookup table	312	3587	114025
Power dissipation (mW)	78	612	35087

Pipelining technique is applied with the help of High-Level Synthesis (HLS) tool [21] which overlaps the execution of statements, increases the overall throughput of the design and reduces the latency as shown in Fig. 17. The default sequential operation is shown in Fig. 17a which requires 8 clock cycles to complete two iterations, if we assumed that each statement needs one clock cycles while in the pipelined version of the loop, shown in Fig. 17b, requires only 5 clock cycles. This leads to improving both the throughput and latency.



a) without loop pipelining b) with loop pipelining
Fig. 17. Loop pipelining technique

Moreover, the dataflow pipelining optimization technique is also applied. This is a very effective technique since it takes a sequential loop and creates parallel process architecture. The steps of Kalman filter can operate in parallel by applying dataflow technique which is supported by HLS [21]. HLS automatically inserts channels between these steps to insure that the data can flow asynchronously from the first statement to the next one as shown in Fig. 18. This results in improving both throughput and latency.



a) without dataflow pipelining b) with dataflow pipelining
Fig. 18. Loop dataflow pipelining technique

The noise covariance (P_k) and Kalman gain (K) are also executed in parallel with prediction and measurement steps since there is no dependency.

V. SIMULATIONS AND RESULTS

VHDL component of Kalman filter, which is optimized by applying different optimization techniques, was synthesized on Xilinx ISE [22] on XC7A100T CSG324-3FPGA device. Table 2 lists the overall performance results in terms of area, power consumption, and maximum frequency for both high and medium precisions. The precision identifies the measurement precision since we used fixed point implementation, high precision (Integer bits=10, fraction bits=20) and medium precision (Integer bits=8, fraction bits=12). The performance is measured with respect to many evaluation metrics; the throughput is given in terms of frequency; hardware utilization is given in terms of the number of slices, Flip Flop

(FF), Lookup table (LUT), BRAM_18K, Digital Signal Processing (DSP48E), and the power dissipation. It is noted that the performance of both directions is the same as in toward radar and far away from the radar because it can be executed in parallel since there is no dependency between them. The hardware resources and computation time for high precision are higher than for the expected medium precision.

TABLE 2
RESOURCES UTILIZATION AND OVERALL IMPLEMENTATION PERFORMANCE ON ARTIX7 (XC7A100T CSG324 -3)

Parameters	Medium Precision			High Precision		
	Toward Radar	Far from Radar	Both	Toward Radar	Far from Radar	Both
Maximum frequency (MHz)	10.246	10.269	10.269	5.959	5.959	5.959
Occupied Slices	2267	1971	2297	4455	4339	4496
Slice LUTs	6956	6904	7384	15818	15724	16295
Slices of FF	1022	1041	1296	1688	1647	2106
Number LUT FF Pairs	7352	7153	7785	16285	16195	16856
DSP48E1s	6	6	8	16	16	24
Number of IOBs	85	85	87	115	115	117
Power Consumption (mW)	1433	1433	1531	2439	2439	2595

To complete the performance evaluation circle and comparison purposes, Kalman filter was coded in C for serial computation. The programs have been executed on a conventional PC powered by a 2.6 GHz i7-3720QM CPU with memory RAM 8.0 GB. The result of the execution times for i7 processor, GPU, and FPGA implementations is summarized in Table 3. The result shows that the performance of FPGA implementations is much better than that of other alternative implementations. The superior performance of the FPGA-based implementations is attributed to the highly paralleled and pipelined architecture.

TABLE 3
PERFORMANCE OF DIFFERENT IMPLEMENTATIONS ON DIFFERENT PLATFORMS

Implementation	Density (Number of targets)		
	25	50	100
i7-3720QM CPU (ms)	31.23	126.7	196.4
Medium Precision/ FPGA (ms)	0.61722	1.293	2.83
High Precision/ FPGA (ms)	1.577	3.255	5.2
GPU (ms)	2.1	3.82	6.15

Results in Table 3 also show the impact of changing the number of targets on the performance. It shows that the system achieves a higher speed when the number of targets increases due to exploiting the GPU and FPGA resources and the available parallelism in the Kalman filter steps.

To efficiently test the performance and accuracy of Kalman filter on both FPGA and GPU implementations, a real input sample data was obtained from Marine Radar project, Communications, Control and Signal Processing laboratory and the University of Toledo [23]. Marine radar was used for the observation of birds and quantification of their activity for a number of years. X-band marine radars with higher resolution are used for bird detection. The radar data is collected using a digitizing card XIR3000B from Russell Technologies. The data collected is processed and parallelized using FPGA and GPU for target detection and tracking. The experimental setup of the entire system is shown in Fig. 19.

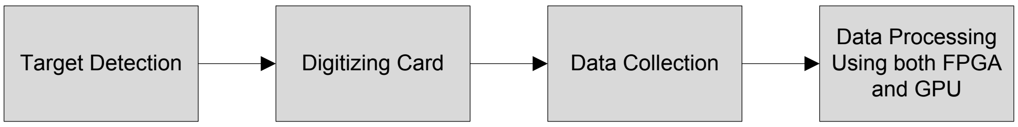


Fig. 19. Experimental setup

The noisy input real data from marine radar was applied and processed using Kalman filter on both FPGA and GPU to examine and verify Kalman filter accuracy and show the performance of the two platforms. Fig. 20 shows the tracking accuracy between the true state vector and the estimated state vector for both FPGA and GPU. GPU architecture is slightly more accurate than FPGA since FPGA is used in fixed point implementation.

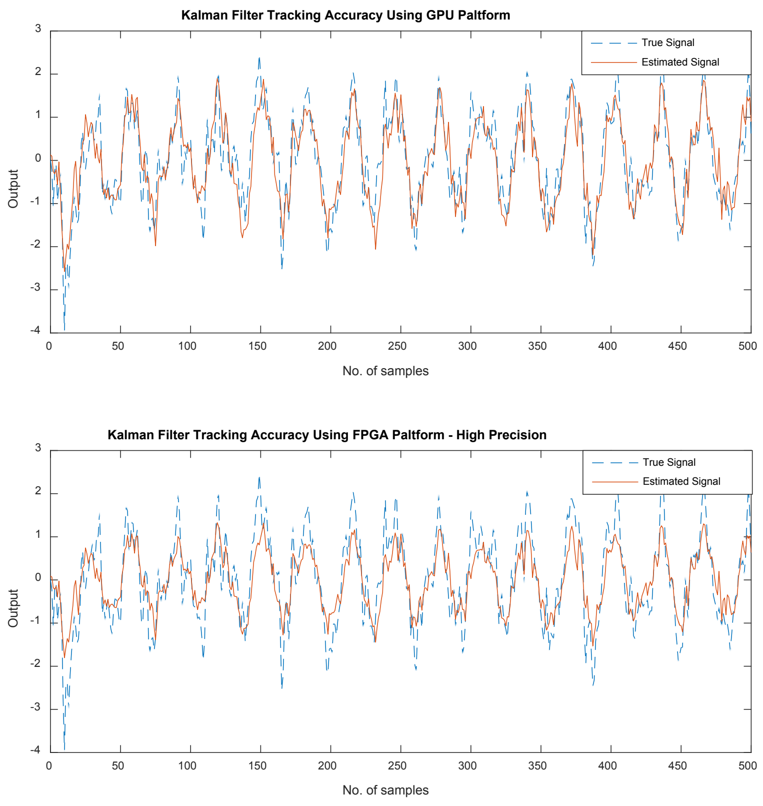


Fig. 20. Tracking accuracy between the true signal and the estimated signal for both FPGA and GPU

In order to show the effectiveness of our implementation, a comparison with other implementations [11] was performed. The implementation in [11] has more power than in our implementation. The comparison in Table 4 shows that our implementation achieves around speed-up 32, while other implementations achieve less than 10.

TABLE 4
OUR IMPLEMENTATION VERSUS OTHER IMPLEMENTATIONS

	Our implementation	Other implementations [12]
Speed-Up	31.93	The best one is 9.1

VI. CONCLUSION

In this paper, an efficient implementation of Kalman filter on both FPGA and GPU platforms is presented. The operations of Kalman filter are decomposed, parallelized, scheduled, and mapped on both platforms. Different optimization techniques for both the computation time and memory utilization are adopted and applied in our model to achieve high performance. Experimental results show the viability of using FPGA and GPU platforms to perform signal processing in real time. The parallel architectures for both FPGA and GPU can significantly outperform an equivalent sequential implementation due to their pipelined architecture, custom functionality of VLSI ASIC devises, flexibility, and adaptability. Our simulation results indicate that the achieved speed-up of FPGA and GPU over the sequential one is improved by up to 37.76 and 31.93, respectively. It is also worth noting that the performance has improved due to increasing input data size. FPGA platform gives a better performance than that of GPU platform.

REFERENCES

[1] J. Gunnarsson, L. Svensson, L. Danielsson and F. Bengtsson, "Tracking vehicles using radar detections," *Proceedings of IEEE Intelligent Vehicles Symposium*, pp. 296-302 , 2007.

[2] S. Blackman and R. Popoli, *Design and Analysis of Modern Tracking Systems*. Norwood, MA: Artech House, 1999.

[3] Y. Bar-Shalom and W. D. Blair, *Multitarget-Multisensor Tracking Volume III: Applications and Advances*, Norwood, MA: Artech House, 2000.

[4] G. Welch and G. Bishop, "An introduction to the Kalman Filter," *SIGGRAPH*, Course 8, 2001.

[5] T. Lacey, *Tutorial: The Kalman Filter*, from: http://mpdc.mae.cornell.edu/Courses/UQ/kf1.pdf

[6] G. Welch and G. Bishop, *An Introduction to the Kalman Filter*, TR 95-041, 2006, from: http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf

[7] D. Simon, *Kalman Filtering, Embedded Systems Programming,* 2007, from: jettronics.de/trac/projects/export/169/.../kalman-dan-simon.pdf

[8] M. I. Ribeiro, *Kalman and Extended Kalman Filters: Concept, Derivation and Properties*, 2004, from: http://users.isr.ist.utl.pt/~mir/pub/kalman.pdf

[9] Z. Merhi, M. Ghantous, M. Elgamel, M. Bayoumi and A. El-Desouki, "A fully-pipelined parallel architecture for kalman tracking filter," *Proceedings of IEEE International Workshop on Computer Architecture for Machine Perception and Sensing*, pp. 81-86, 2006.

[10] T. Blattner and S. Yang, "Performance study on cuda gpus for parallelizing the local ensemble transformed kalman filter algorithm," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 2, pp. 167-177, 2012.

[11] M. Y. Huang, S. C. Wei, B. Huang and Y. L. Chang, "Accelerating the kalman filter on a gpu," *Proceedings of IEEE Parallel and Distributed Systems International Conference*, pp. 1016-1020, 2011.

[12] O. Rosén and A. Medvedev, "Efficient parallel implementation of a kalman filter for single output systems on multicore computational platforms," *Proceedings of IEEE Decision and Control and European Control Conference*, pp. 3178-3183, 2011.

[13] C. Liu, "cuHMM: a CUDA implementation of hidden markov model training and classification," *The Chronicle of Higher Education*, pp. 1-13, 2009.

[14] J. Nielsen and A. Sand, "Algorithms for a parallel implementation of hidden markov models with a small state space," *Proceedings of IEEE Parallel and Distributed Processing Workshops and PhD Forum*, pp. 452-459, 2011.

[15] M. Harris, "Optimizing parallel reduction in CUDA," *NVIDIA Developer Technology*, vol. 2, no. 4, 2007.

[16] R. Kalman, "A new approach to linear filtering and prediction problems," *Journal of Basic Engineering*, vol. 82, pp. 95-108, 1960.

[17] Nvidia Corporation Geforce GTX 260, from: http://www.nvidia.com/object/product_geforce_gtx_260_us.html.

[18] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers, 2001.

[19] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[20] M. Harris, "Optimizing CUDA," *Proceedings of International Conference on High Performance Computing, Networking and Storage*, 2007.

[21] High-Level Synthesis Vivado, from Xilinx, Inc. From: http://www.xilinx.com.

[22] Xilinx Corporartion. 2002, from: www.xilinx.com.

[23] Communications, Control and Signal Processing, from: http://www.eng.utoledo.edu/eecs/research/groups/com_and_sig_proc.html.