EE5332 Final Course Project:
A study of Hardware Acceleration of Kalman Filter

Saurav Sachin Kale (EE19B141), Surya Prasad S (EE19B121), Arun Krishna AMS (EE19B001)

May 20, 2022

Part I

The Kalman Filter

1 The Algorithm

Kalman Filter is an algorithm that produces estimates of state variables of a discrete data controlled system (thus an estimate of the state of the system) based on measurements which can typically be noisy.
We have the following parameters in a Kalman Filter:

- $\vec{x_k}$: state vector
- $\vec{u_k}$: control-input vector
- $\vec{w_k}$: process noise vector, assumed to be Gaussian with zero mean and covariance Q
- $\vec{z_k}$: measurement vector
- $\vec{\nu_k}$: measurement noise vector
- F : state transition matrix
- B : control-input matrix
- H : measurement matrix, assumed to be Gaussian with zero mean and covariance R

Together, F, B, H, Q, and R together define the system. The job of the Kalman filter is to produce estimates of $\vec{x_k}$given the initial estimate x_0, series of measurements z_k along with the system definition.
The evolution of state parameters from $k-1^{th}$ iteration to the k^{th} iteration (process model) is:

$$\vec{x_k} = F\vec{x_{k-1}} + B\vec{u_{k-1}} + \vec{w_{k-1}}$$

The measurement model is:

$$\vec{z_k} = H\vec{x_k} + \vec{\nu_k}$$

Presented below is the algorithm of Kalman Filter which produces estimates for $\vec{x}$ every iteration:

Algorithm 1 Kalman Filter Algorithm

Prediction:

$$\vec{x_k^-} = F\vec{x_{k-1}^+} + B\vec{u_{k-1}}$$

$$P_k^- = FP_{k-1}^+F^T + Q$$

Update:

$$\tilde{y_k} = \vec{z_k} - H\vec{x_k^-}$$

$$K_k = P_k^-H^T(R + HP_k^-H^T)^{-1}$$

$$\vec{x_k^+} = \vec{x_k^-} + K_k\tilde{y_k}$$

$$P_k^+ = (I - K_kH)P_k^-$$

Where

- P_k : error state covariance
- $\tilde{y_k}$: measurement residual
- K_k : Kalman gain
- Superscript of $+$ denotes post update values and $-$ denotes pre update values in an iteration

2 Analysis and optimizations

For an n state variables, n input variables and n measurements Kalman Filter, the following . Suffixes imply operations:

- M: Multiply
- A: add/sub
- I: Inverse

Step	Estimated Operations
$\vec{x_k^-} = F\vec{x_{k-1}^+} + B\vec{u_{k-1}}$	n^2M+$n(n-1)$A+n^2M+$n(n-1)$A+nA
P$_k^- = FP_{k-1}^+F^T + Q$	n^3M+$n^2(n-1)$A+n^3M+$n^2(n-1)$A+n^2A
$\tilde{y_k} = \vec{z_k} - H\vec{x_k^-}$	n^2M+$n(n-1)$A+nA
K$_k = P_k^- H^T(R + HP_k^- H^T)^{-1}$	n^3M+$n^2(n-1)$A+n^3M+$n^2(n-1)$A+n^2A+$\approx n^3$I
$\vec{x_k^+} = \vec{x_k^-} + K_k\tilde{y_k}$	n^2M+$n(n-1)$A+nA
P$_k^+ = (I - K_kH)P_k^-$	n^3M+$n^2(n-1)$A+n^2A+n^3M+$n^2(n-1)$A

The total order of operations comes to about $13n^3$, in the range of 2-25 KFLOPs for typical Kalman Filter sizes (6, 12 etc). However, one must note that for realtime control applications, the iteration interval should be in the range of ms. This implies a throughput of typically 5-10 MFLOPS. The data rate involves feeding $x_k, u_k, P_k, z_k = 54$ floating point numbers every iteration, therefore

$$\frac{54 \times 4 \times 8b}{1ms} = 1.7Mbps$$

So we have

- Bitrate: 1.7Mbps
- Throughput: 5-10 MFLOPS

A few approaches come to mind in order to implement the Kalman Filter:

- CPU Single Threaded: Would be implemented in C/C++. It is easy to implement, and can be used as a baseline to compare implementations
- CPU Multithreaded: This approach also seems feasible, but there is limited parallelism in the algorithm and typical Kalman filters are not large in size
- GPU: Theoretically this would be easily capable of speeding up matrix operations but the cost of synchronisation and memory transfer is dominant for small matrices
- Custom Hardware for FPGA: The merits of this could be cost, lower power consumption and could be ideal for deploying low-end low-power systems. Several application specific and general optimizations like fixed point could be made to drastically lower the latency of operations.

We tried three of the above approaches:

- Baseline C
- CPU Multithreading using OpenMP
- Custom Hardware description using Bluespec SystemVerilog and using Vitis HLS

Analysing the Directed Acyclic Graph (DAG) of this algorithm presents some interesting insights on possible optimizations:

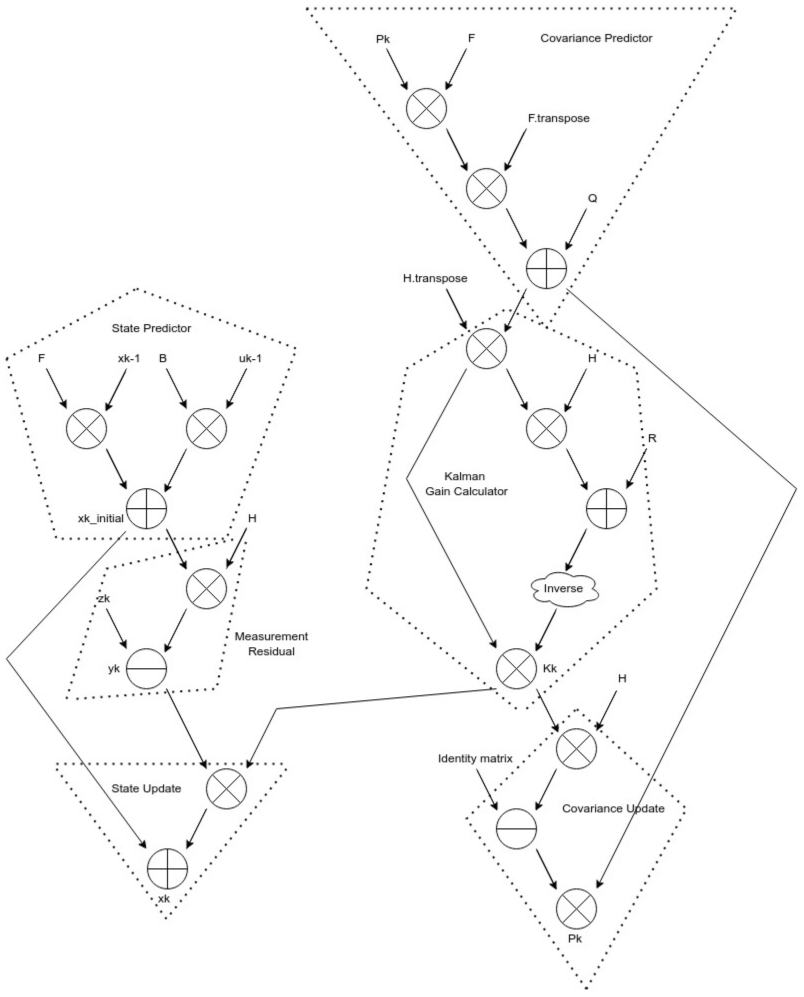


Figure 1: DAG for Kalman Iteration (direction of control flow is downwards)

The direction of dataflow is downwards. At every iteration, the covariance matrix Pk and the state vector Xk are the outputs of the system, with the measurement zk, input vector uk, state vector Xk and covariance matrix Pk taken as inputs. From the given graph we can see that the state predictor and measurement residual equations (the tree on the left) is independent of the Covariance predictor and kalman gain calculator. Thus these can be parallelized. For the remainder of this report, we shall refer to the state predictor and measurement residual of the DAG as the left branch and the covariance predictor and kalman gain calculator as the right branch.

Part II

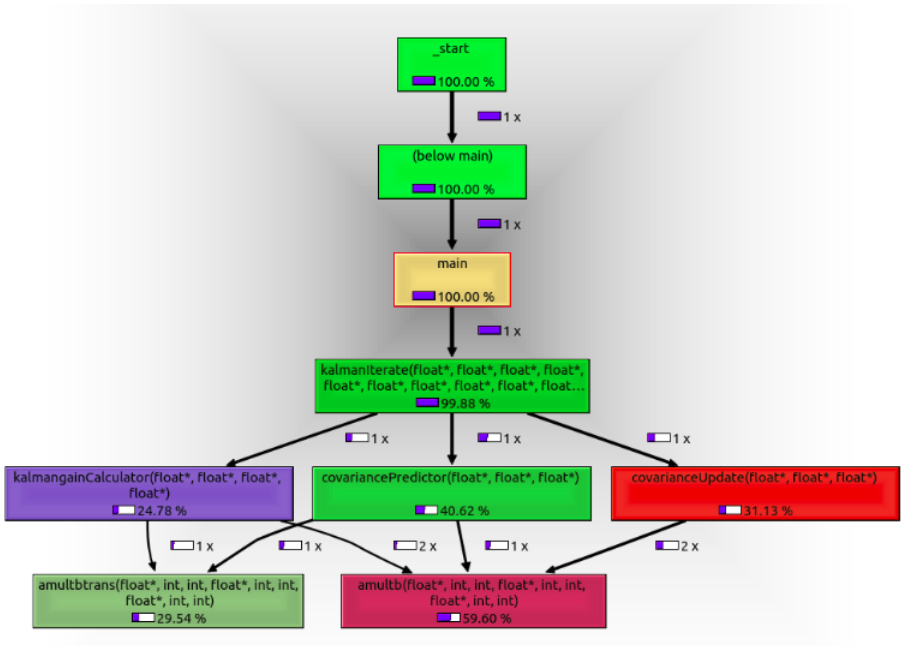
Implementation and results

1 Baseline: Single threaded C

Single threaded Kalman filter was implemented in C. Profiling tools like gprof, Valgrind and google-preformance tools were used to analyze the C-implementation. gprof is a sampling based profiling tool. gprof and google-performance tools did not give useful results.

Valgrind gave the most accurate results. Callgrind - Valgrind's tool is used to profile the code and the output is visualized through KCachegrind tool.
The test case consisted of a 12 dimensional state variable vector, 6 dimensional measurement vector and 6 dimensional input vector.

One would expect, matrix inverse to be costlier, but it is observed from the graph that matrix multiplications take the largest amount of time. This is because the number of matrix multiplication operations (10 matrix multiplications) are significantly



_start
100.00 %
1 x
(below main)
100.00 %
1 x
main
100.00 %
1 x
kalmanIterate(float*, float*, float*, float*, float*, float*, float*, float*, float*, float...
99.88 %
1 x
1 x
1 x
kalmangainCalculator(float*, float*, float*, float*)
24.78 %
covariancePredictor(float*, float*, float*)
40.62 %
covarianceUpdate(float*, float*, float*)
31.13 %
1 x
1 x
2 x
1 x
2 x
amultbtrans(float*, int, int, float*, int, int, float*, int, int)
29.54 %
amultb(float*, int, int, float*, int, int, float*, int, int)
59.60 %

Figure 2: Valgrind's analysis output

higher compared to matrix inversion (only one matrix inversion).

We can see that the Kalmangain_Calculator, Covariance_predictor, Covariance_Update have the maximum computational complexity. These three functions should be parallelized for lower latencies, but since the functions are dependent on each other - they have to be executed sequentially.

2 Multithreading in CPU using OpenMP

From the DAG in Figure 1, we try to exploit the parallelism offered by the branches. We used OpenMP to parallelise operations. Our initial attempt was to parallelise Matrix*Matrix operations and Matrix*Vector operations but the small sizes of the operands (6, 12) led to the synchronisation overheads being very significant. We then tried to implement the Kalman filter by having task parallelism using OpenMP. Task parallelism focuses on distributing tasks—concurrently performed by processes or threads—across different processors. Based on the SFG and the Valgrind profiling result, we parallelized the KalmanGain Calculator and the Covariance Predictor in one task and the State Predictor and Measurement Residual as another task.

The number of instructions for both implementations for each iteration is around the same ~420000 instructions. So there were no costs of memory due to parallelisation.

When we timed the implementation for 100 iterations:

- Single threaded 100 cycles: 0.014 seconds
- Task Parallelized 100 cycles: 0.00568 seconds
- Single threaded 1 cycle: 0.00014 seconds
- Task Parallelized 1 cycle: 0.0000568 seconds

As we can see the task parallelized implementation is $\sim$ 20 times faster compared to single threaded implementation.

3 Custom Hardware Description in Bluespec SystemVerilog

For custom hardware we decided to go with Bluespec SystemVerilog over Verilog due to the following reasons:

- Ease of use: Can directly map the task schedule to hardware description using Bluespec's rule based approach, easier to write sophisticated state machines
- More control: Closer to hardware description than Vitis HLS, more control over architecture, clock cycle counts and latency
- Library support: We made use of the FixedPoint library of Bluespec. We have used the configuration FixedPoint<16, 16>

There were two major operations which had to be dealt with, namely matrix multiplication and matrix inversion. Other modules included vector dot product which was also completely pipelined. From the DAG in Figure 1, we can see that there are a large number of Matrix*Matrix operations scheduled in series which is computed parallel to the Matrix*Vector operations. So we decided to use one systolic Matrix multiplier for all the Matrix*Matrix operations. We can afford to give Matrix*Vector more latency and hence we went with a simple pipelined Vector Dot product module. We went with 2 vector dot product modules because of the two parallel operations in State Predictor function. We chose to directly add the elements in a single cycle for add operations between matrices and vectors.

Additionally, our hardware is extensible for any size of inputs. This flexibility is hard to achieve in Verilog and doesn't provide consistent results in HLS.

3.1 Matrix Multiplication

We went with a systolic array based approach with multiply accumulate processing elements. The input and output of this module are of the length of a vector. This reduces the number of ports and also allows for pipelining, although we were not able to pipeline it as we ran out of time. Currently the top module issues each matrix multiplication sequentially.

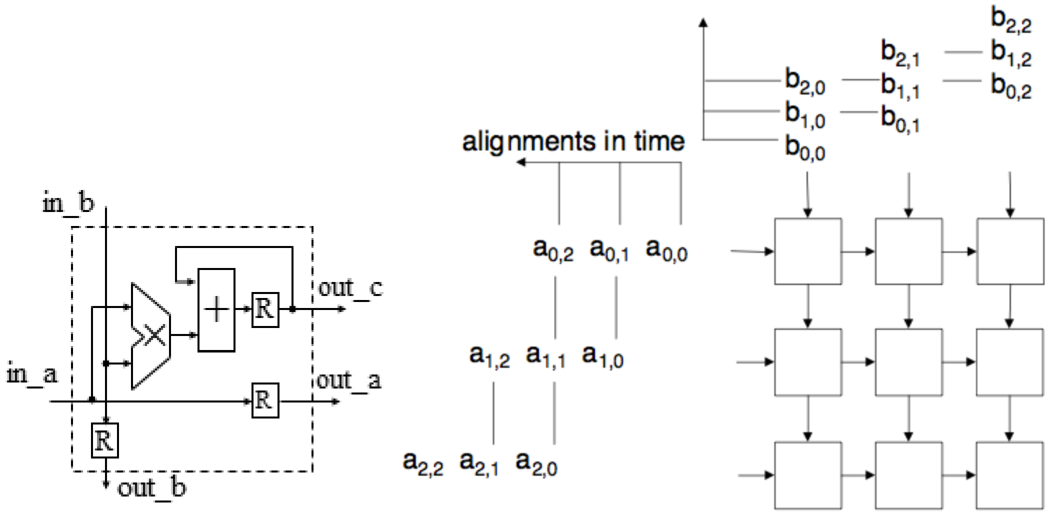

in_b
out_c
in_a
out_a
out_b
R
alignments in time
$b_{2,2}$ $b_{2,1}$ $b_{1,2}$ $b_{2,0}$ $b_{1,1}$ $b_{0,2}$ $b_{1,0}$ $b_{0,1}$ $b_{0,0}$
$a_{0,2}$ $a_{0,1}$ $a_{0,0}$
$a_{1,2}$ $a_{1,1}$ $a_{1,0}$
$a_{2,2}$ $a_{2,1}$ $a_{2,0}$

Figure 3: Architecture for matrix multiplication systolic array

This resulted in an architecture that takes $3n - 2$ clock cycles to compute the product. A significant improvement from the $O(n^3)$ operations estimated, even if we take best case 1 clock per operation.

3.2 Matrix Inverse

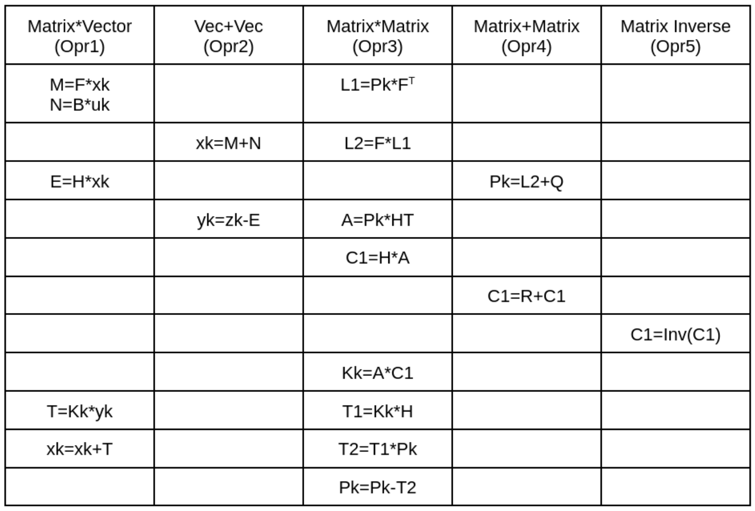
Several architectures were explored (such as QR and LU decomposition) but we ended up going with Gauss Jordan elimination using row operations, as it seemed to be easiest to implement and not a lot of latency was sacrificed. For a non-singular matrix M of dimensions $n \times n$, the algorithm works in two stages:

- In the first n clock cycles, each column is made a pivot by the operation $R_k \leftarrow R_k - \alpha R_p$ where R_k is any row except pivot row, and R_p is the pivot row. α is the ratio between $R_k[p]$, the element in the same column but in R_k and the pivot element. This, done for all other rows at the same time, ensures one column becomes a pivot every cycle. Thus in n cycles, all the columns become pivots.
- A complication can occur if the pivot element is 0, we get a divide by zero in computation of α. In that case, we perform the following operation: $R_p \leftarrow R_p + \sum R_k$ such that $k > p$ (add all the rows below and including R_p). This will only fail if the matrix is singular, therefore it solves our problem.
- Finally, each row is divided by the value of the pivot element.

The above operations are applied to the input matrix M and the identity matrix (which we shall call the output matrix N) $I_{n\times n}$. By the end, M becomes $I_{n\times n}$ and N becomes M^{-1}. This results in a latency of $n + 1$ clock cycles for the inverse. A significant improvement over the $O(n^3)$ operations estimated.

3.3 Top module

With the elementary operations defined, a state machine was created which applied the relevant operations one by one to the inputs to produce the estimates of state variables.



Matrix*Vector (Opr1)	Vec+Vec (Opr2)	Matrix*Matrix (Opr3)	Matrix+Matrix (Opr4)	Matrix Inverse (Opr5)

M=F*xk
N=B*uk
		L1=Pk*F^T		

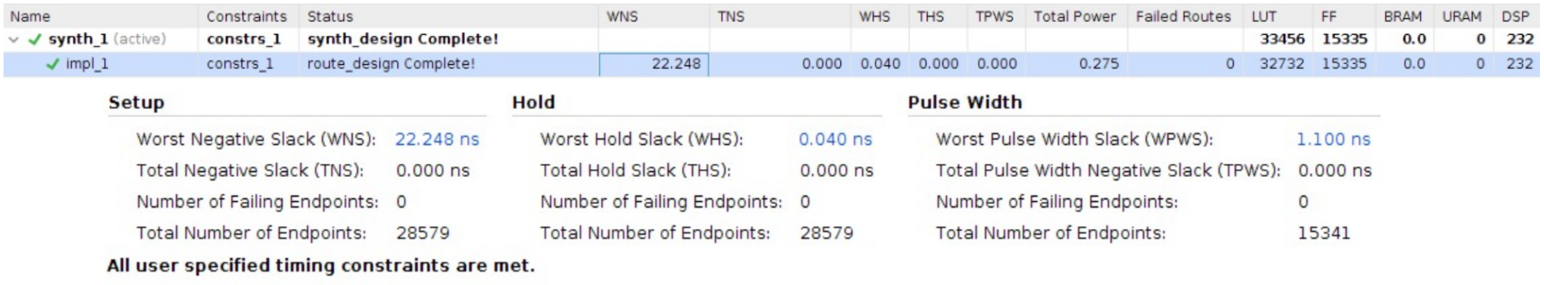
	xk=M+N	L2=F*L1		
E=H*xk			Pk=L2+Q	
	yk=zk-E	A=Pk*HT		
		C1=H*A		
			C1=R+C1	
				C1=Inv(C1)
		Kk=A*C1		
T=Kk*yk		T1=Kk*H		
xk=xk+T		T2=T1*Pk		
		Pk=Pk-T2		

Figure 4: Task schedule for the top module

The rules in the top module are based on the functions used in the C code and are scheduled according to the table above.

3.4 Results

We were able to get a single iteration Kalman filter outputs x_k and P_k every 198 clock cycles. One could pipeline this workflow and keep extracting a new estimate for x_k in around 150 cycles, however we could not do that due to lack of time. 198 clock cycles is a significant improvement over the estimated $O(n^3)$ operations. This is because of the use of parallelism and various optimal architectures described above.
A quick synthesis of the generated verilog in Vivado gave the following results:



Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
✓ synth_1 (active)	constrs_1	synth_design Complete!								33456	15335	0.0	0	232
✓ impl_1	constrs_1	route_design Complete!	22.248	0.000	0.040	0.000	0.000	0.275	0	32732	15335	0.0	0	232

Setup

Worst Negative Slack (WNS):	22.248 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	28579

Hold

Worst Hold Slack (WHS):	0.040 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	28579

Pulse Width

Worst Pulse Width Slack (WPWS):	1.100 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	15341

All user specified timing constraints are met.

Figure 5: Vivado Synthesis results of Bluespec Design

This was for a clock of 7MHz. Our negative slack is 22.248ns. This implies our maximum clock frequency is

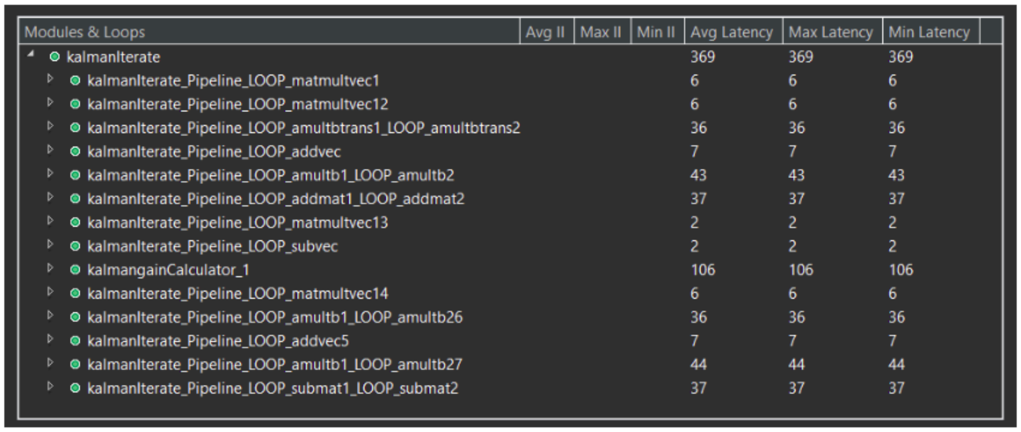
$$\frac{1}{\frac{1}{7\times10^6} - 22.248 \times 10^{-9}} = 8.29MHz$$

This means a maximum latency of 0.00002388 sec per set of outputs, which is a speedup of

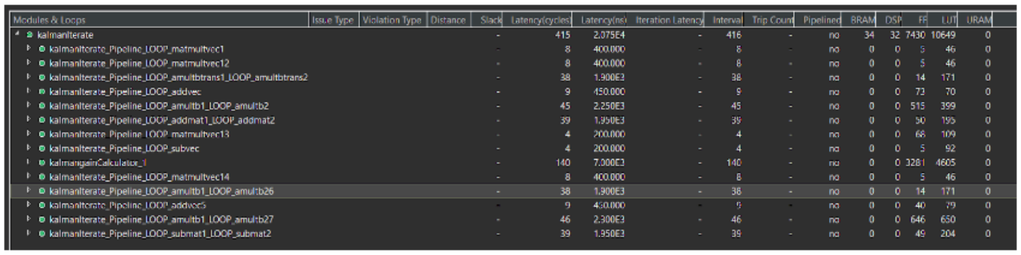
$$\frac{0.00014}{0.00002388} = 5.8$$

4 Hardware Design using Vitis HLS

The baseline C-code initially synthesized with minimal optimizations to estimate the resource utilization of the baseline. This was further optimized and was used in the synthesis of hardware using Vitis HLS.



Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kalmanIterate				369	369	369
kalmanIterate_Pipeline_LOOP_matmultvec1				6	6	6
kalmanIterate_Pipeline_LOOP_matmultvec12				6	6	6
kalmanIterate_Pipeline_LOOP_amultbtrans1_LOOP_amultbtrans2				36	36	36
kalmanIterate_Pipeline_LOOP_addvec				7	7	7
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb2				43	43	43
kalmanIterate_Pipeline_LOOP_addmat1_LOOP_addmat2				37	37	37
kalmanIterate_Pipeline_LOOP_matmultvec13				2	2	2
kalmanIterate_Pipeline_LOOP_subvec				2	2	2
kalmangainCalculator_1				106	106	106
kalmanIterate_Pipeline_LOOP_matmultvec14				6	6	6
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb26				36	36	36
kalmanIterate_Pipeline_LOOP_addvec5				7	7	7
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb27				44	44	44
kalmanIterate_Pipeline_LOOP_submat1_LOOP_submat2				37	37	37



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kalmanIterate				-	415	2.075E4	-	416	-	no	34	32	7430	10649	0
kalmanIterate_Pipeline_LOOP_matmultvec1				-	8	400.000	-	8	-	no	0	0	5	46	0
kalmanIterate_Pipeline_LOOP_matmultvec12				-	8	400.000	-	8	-	no	0	0	5	46	0
kalmanIterate_Pipeline_LOOP_amultbtrans1_LOOP_amultbtrans2				-	38	1.900E3	-	38	-	no	0	0	14	171	0
kalmanIterate_Pipeline_LOOP_addvec				-	9	450.000	-	9	-	no	0	0	73	70	0
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb2				-	45	2.250E3	-	45	-	no	0	0	515	399	0
kalmanIterate_Pipeline_LOOP_addmat1_LOOP_addmat2				-	39	1.950E3	-	39	-	no	0	0	50	195	0
kalmanIterate_Pipeline_LOOP_matmultvec13				-	4	200.000	-	4	-	no	0	0	68	109	0
kalmanIterate_Pipeline_LOOP_subvec				-	4	200.000	-	4	-	no	0	0	5	92	0
kalmangainCalculator_1				-	140	7.000E3	-	140	-	no	0	0	3281	4605	0
kalmanIterate_Pipeline_LOOP_matmultvec14				-	8	400.000	-	8	-	no	0	0	5	46	0
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb26				-	38	1.900E3	-	38	-	no	0	0	14	171	0
kalmanIterate_Pipeline_LOOP_addvec5				-	9	450.000	-	9	-	no	0	0	40	79	0
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb27				-	46	2.300E3	-	46	-	no	0	0	646	650	0
kalmanIterate_Pipeline_LOOP_submat1_LOOP_submat2				-	39	1.950E3	-	39	-	no	0	0	49	204	0

Figure 6: Baseline C synthesis results with minimal optimizations

- C Synthesis Expected Latency: 415 cycles
- C/RTL Cosimulation Latency: 369 cycles
- Estimated frequency max based on C synthesis: 31.62 MHz

This alone provided a latency of 0.00001167 gives a 11.997x speedup. Several optimizations were made after this.

Fixed point operations with ap_fixed<16,16> were explored in the early stages of the project. Significant challenges were faced especially in the matrix inversion operation because of the division operation. Hence, floating point operations were used.

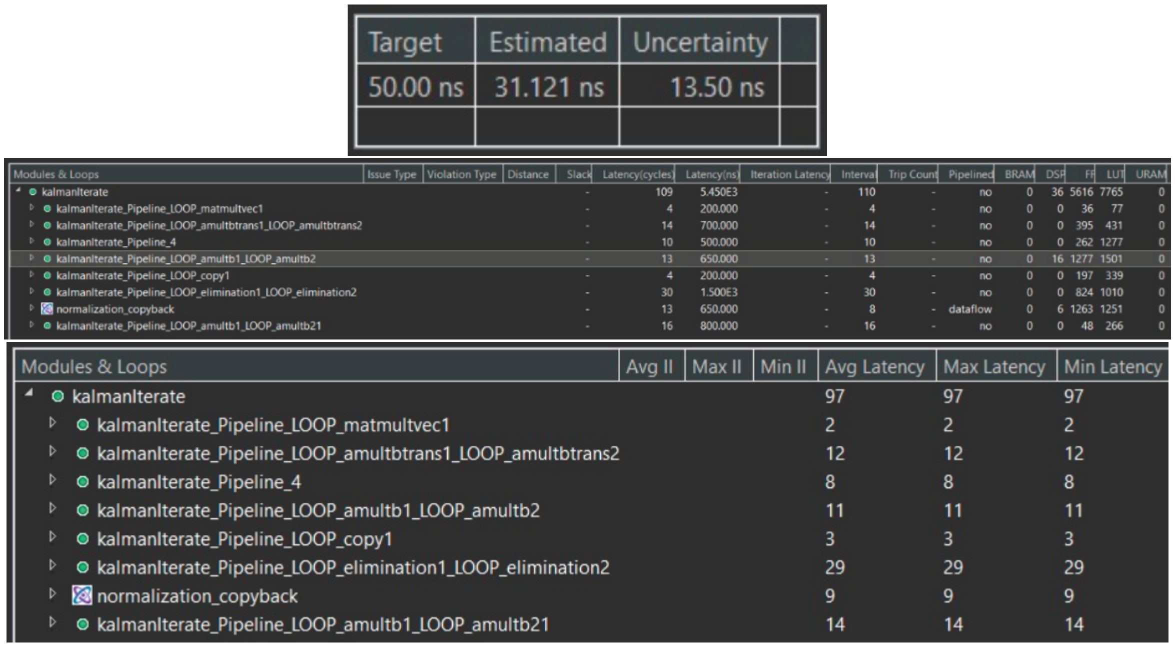
The matrix and the vectors were initially implemented in BRAMs. Since the number of access ports - read/write were just two, significant number of cycles were spent in accessing the data. This significantly reduced the hardware utilization efficiency and prevented further unrolling. To solve this problem, the array was completely partitioned into individual registers. The complete partitioning is viable for the dimensions we considered (6 state variables, 6 input variables and 2 measurement variables). This allowed us to unroll to a very large extent, thereby providing a very high speed-up.

The addition and subtraction of matrices and vectors were completely unrolled. Since matrix multiplication and division operations takes up significant resources, they were partially unrolled (Unrolled by a factor of number of columns).

Functions are inlined to reduce the overhead while jumping between functions. Loops were made Perfect or Semi-Perfect (inner loops always had fixed number of iterations & only inner loop containing the body) to flatten them - thereby reducing the enter & exit overhead (reduces number of clock cycles by 2N where N is the number of iterations in the outer loop).

Several design choices were made during the design of matrix inverse. One such choice that was made after careful analysis was to add the rows instead of exchange of rows when the pivot element is zero - This resulted in couple of clock cycles.

In our previous submission, we concentrated on reducing the number of clock cycles in inverse operation. In the last couple of days, our focus shifted towards other operations(addition & subtraction of matrices) which resulted in drastic reduction in resources and latency.



Target	Estimated	Uncertainty
50.00 ns	31.121 ns	13.50 ns

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kalmanIterate				-	109	5.450E3	-	110	-	no	0	36	5616	7765	0
kalmanIterate_Pipeline_LOOP_matmultvec1				-	4	200.000	-	4	-	no	0	0	36	77	0
kalmanIterate_Pipeline_LOOP_amultbtrans1_LOOP_amultbtrans2				-	14	700.000	-	14	-	no	0	0	395	431	0
kalmanIterate_Pipeline_4				-	10	500.000	-	10	-	no	0	0	262	1277	0
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb2				-	13	650.000	-	13	-	no	0	16	1277	1501	0
kalmanIterate_Pipeline_LOOP_copy1				-	4	200.000	-	4	-	no	0	0	197	339	0
kalmanIterate_Pipeline_LOOP_elimination1_LOOP_elimination2				-	30	1.500E3	-	30	-	no	0	0	824	1010	0
normalization_copyback				-	13	650.000	-	8	-	dataflow	0	6	1263	1251	0
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb21				-	16	800.000	-	16	-	no	0	0	48	266	0

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
kalmanIterate				97	97	97
kalmanIterate_Pipeline_LOOP_matmultvec1				2	2	2
kalmanIterate_Pipeline_LOOP_amultbtrans1_LOOP_amultbtrans2				12	12	12
kalmanIterate_Pipeline_4				8	8	8
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb2				11	11	11
kalmanIterate_Pipeline_LOOP_copy1				3	3	3
kalmanIterate_Pipeline_LOOP_elimination1_LOOP_elimination2				29	29	29
normalization_copyback				9	9	9
kalmanIterate_Pipeline_LOOP_amultb1_LOOP_amultb21				14	14	14

Figure 7: C-Synthesis result and RTL Co-Simulation Tool result

We were able to get the latency to 97 clock cyles with a 20 MHz clock, with a maximum possible clock frequency of ~32 MHz. This results in one set of outputs being produced at a minimum of 0.0000030312 sec, which is a speedup of

$$\frac{0.00014}{0.0000030312} = 46.186$$

Part III
Conclusion

1 Comparison of implementations

Apart from the C baseline, all three implementations are suitable for the requirements of realtime control applications where a Kalman Filter could be used (till 12x12 dimensions). Across implementations however, we find that Vitis HLS provides the fastest results, and is an ideal tool for such DSP applications. Bluespec implementation offers more control, however it is extremely complex and time consuming, and the results one gets may not be as fast unless a lot of lower level optimizations are exploited.

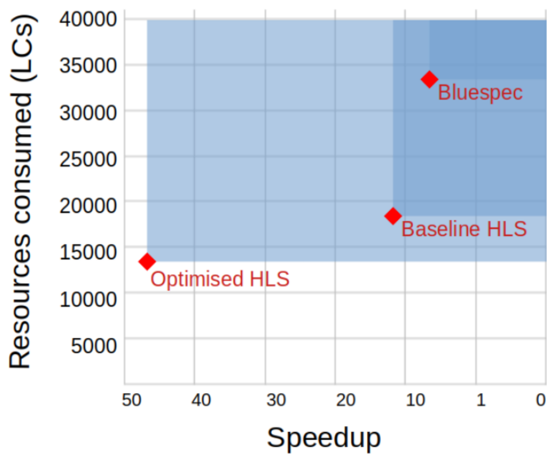


Figure 8: Pareto Optimality chart of implementations

2 Future scope

There were several things we could not do due to lack of time and the immense complexity of designs.

- Analyse the implementations for even higher dimensions. Higher dimensions are suitable for nonlinear control applications
- Analyse the HLS design output and make improvements in the Bluespec code based on it
- Reduce the Bluespec resource utilization

Part IV
Work distribution and References

Work Distribution

Code is available at this link. https://drive.google.com/drive/folders/1vISZqeXRpmLeS67UfpNO1wSmEEzFgNsv

- Baseline C: AMS, Saurav
- OpenMP: Surya
- Bluespec (mat-mul, mat-inv, and other operations): Saurav
- Bluespec (top module): Saurav, Surya
- Analysis: Saurav, AMS
- Vitis HLS: AMS, Surya
- Profiling: AMS

References

[1] Amin Jarrah, Abdel-Karim Al-Tamimi and Tala Albashir, "Optimized Parallel Implementation of Extended Kalman Filter Using FPGA", Journal of Circuits, Systems, and Computers Vol. 27, No. 1 (2018)

[2] C.R. Lee, Z. Salcic, "High-performance FPGA-based implementation of Kalman filter", Microprocessors and Microsystems 21 (1997) 257-265

[3] Ashan's Blog http://ashanpeiris.blogspot.com/2015/08/digital-design-of-systolic-array.html

[4] Vivado HLS Optimization Methodology Guide, Xilinx ver. 2018