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 17, 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. It is easy to implement, and can be used as a baseline to compare implementations
- GPU: Theoretically this would be easily capable of such a workflow from the throughput and bitrate
- CPU Multithreaded: This approach also seems feasible, since typical Kalman filters are not large in size
- Custom Hardware: This would include an FPGA design. The merits of this could be cost, lower power consumption and could be ideal for deploying for 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 shall try three such approaches:

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

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

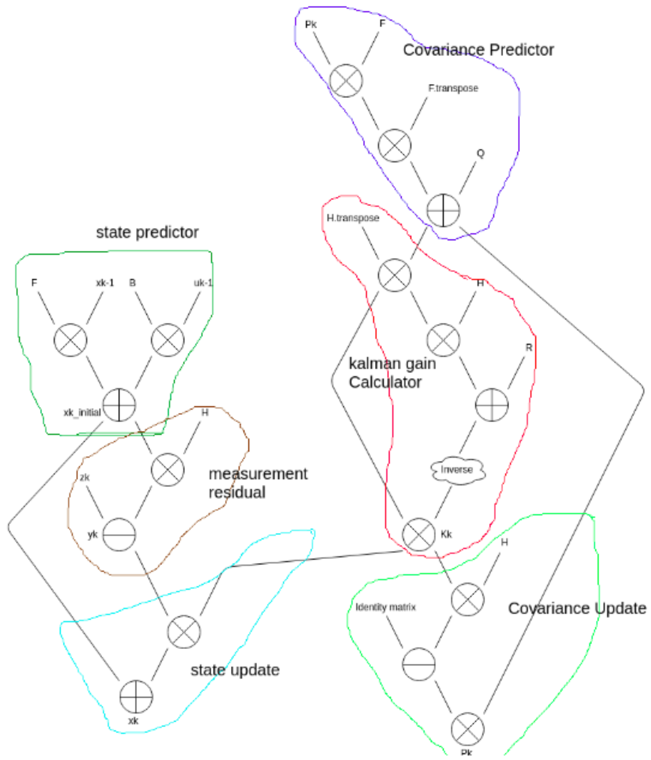


Figure 1: DAG for Kalman Iteration (direction of control flow is downwards)

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.

Part II

Implementation and results

1 Baseline: Single threaded C

A working C++ code implementation of Kalman filter which uses "Eigen" matrix library was implemented. The "Eigen" library is optimized for large matrices – but has huge overhead. But considering that the dimensions of matrices generally involved in the Kalman filters are less than 20 (It depends a lot on applications. But general use-cases like localization and sensor fusion in mobile robots have operations on matrices with dimensions less than 15) - we realize that overhead is way high for any optimization to be useful. Hence all the matrix operations (multiplication, inverses) were implemented in C directly (Single threaded). Inverse of matrix is implemented using Gauss-Jordan Elimination method.

Profiling tools like gprof, Valgrind and google-preformance tools were used to analyze the C-implementation. gprof is a sampling based profiling tool. Since the execution of functions (that describe individual equations) was really fast and because of the low sampling frequency(100Hz) of gprof, no useful information was obtained through gprof tool. Google-perf tools were used with multiple sampling frequencies, but it never worked.

Out of all three profiling tools - Valgrind gave the most accurate results. Valgrind runs the program in Valgrind virtual machine. Callgrind - Valgrind's tool is used to profile the code. Since Valgrind is a simulated virtual machine, it gives the best accurate results but at the same time its extremely slow. Callgrind's output is visualized through KCachegrind tool.
The test case considered consisted of a 12 dimensional state variable vector, 6 dimensional measurement vector and 6 dimensional input vector. We obtained these counts for number of instructions in both approaches:

With Eigen	From scratch
2,315,849 instructions	569,774 instructions

As expected it has huge overhead. But considering the C-code needs to be implemented in Vitis HLS - we proceed our analysis with the custom-built version.

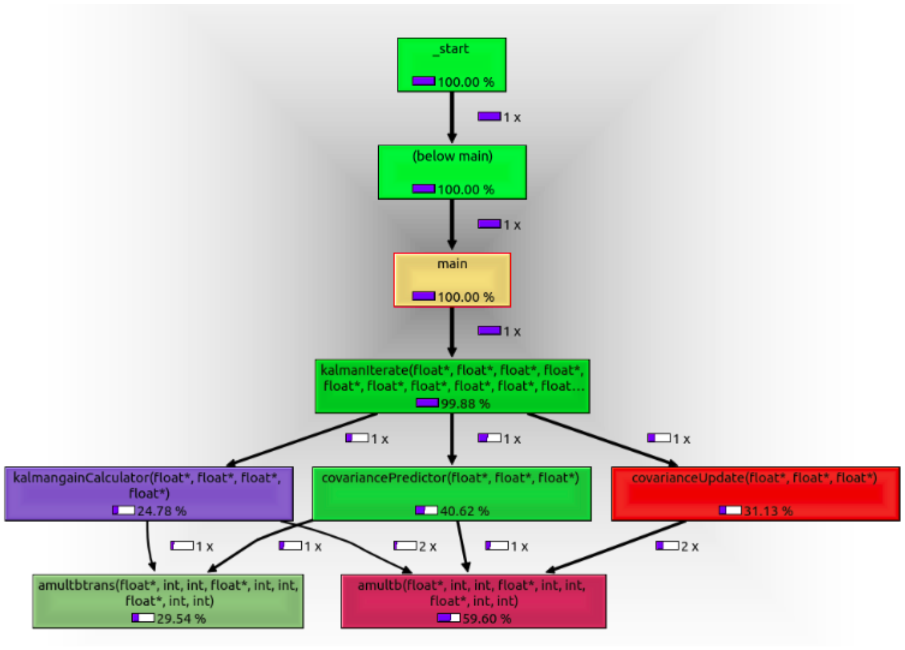

_start
100.00 %
1 x
(below main)
100.00 %
1 x
main
100.00 %
1 x
kalmanIterate(float*, float*, float*, float*, float*, float*, float*, float*, float*, float...
99.88 %
1 x
1 x
1 x
kalmangainCalculator(float*, float*, float*, float*)
24.78 %
covariancePredictor(float*, float*, float*)
40.62 %
covarianceUpdate(float*, 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

2 Multithreading in CPU using OpenMP

We now try 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 parallelize the following KalmanGain Calculator and Covariance Predictor as a single task State Predictor and Measurement Residual as a single task.

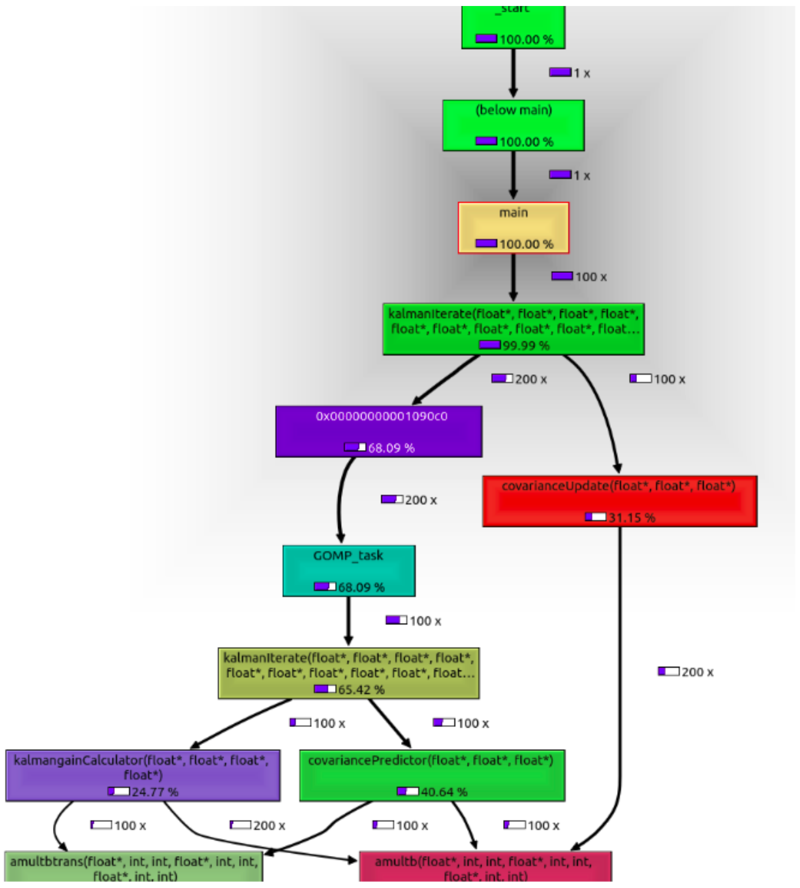

_start
100.00 %
1 x
(below main)
100.00 %
1 x
main
100.00 %
100 x
kalmanIterate(float*, float*, float*, float*, float*, float*, float*, float*, float*, float...
99.99 %
200 x
100 x
0x00000000001090c0
68.09 %
200 x
covarianceUpdate(float*, float*, float*)
31.15 %
GOMP_task
68.09 %
100 x
200 x
kalmanIterate(float*, float*, float*, float*, float*, float*, float*, float*, float*, float...
65.42 %
100 x
100 x
kalmangainCalculator(float*, float*, float*, float*)
24.77 %
covariancePredictor(float*, float*, float*)
40.64 %
100 x
200 x
100 x
100 x
amultbtrans(float*, int, int, float*, int, int, float*, int, int)
amultb(float*, int, int, float*, int, int, float*, int, int)

Figure 3: Valgrind profiling result

The number of instructions for both implementations for each iteration is around the same ~420000 instructions.

When we time 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 ~ 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 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 clock cycle counts and latency
- Library support: We made use of the FixedPoint library which was readily available by default. In the entire bluespec implementation, we have used 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 can afford to give Matrix*Vector more latency. Hence we went with just 2 vector dot product modules to compute the result for Matrix*Vector operations. We chose to directly add the elements in a single cycle for add operations between matrices and vectors.

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.

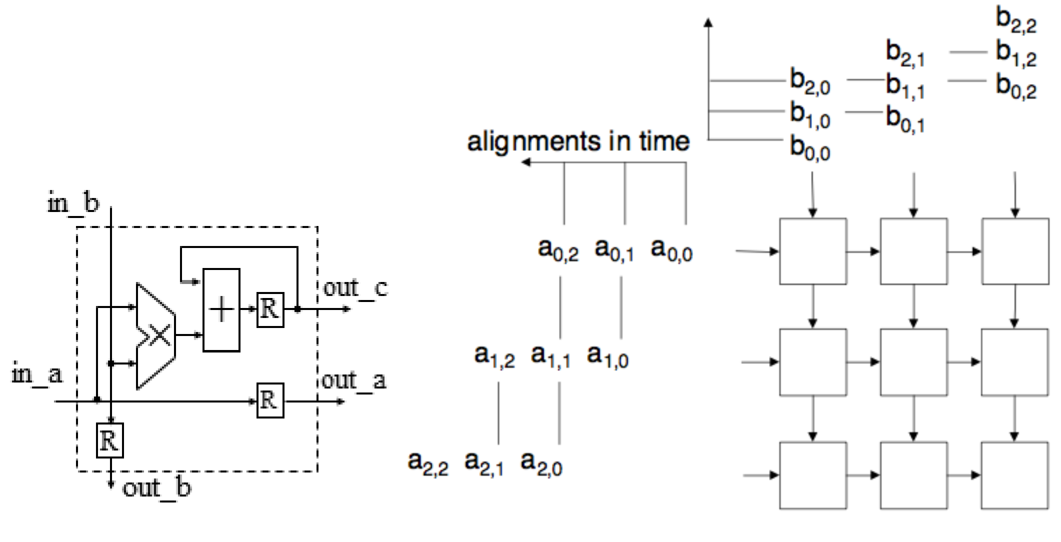

in_b
out_c
in_a
out_a
out_b
R
alignments in time
$b_{2,2}$ $b_{2,1}$ $b_{1,2}$ $b_{2,0}$ $b_{1,1}$ $b_{0,2}$ $b_{1,0}$ $b_{0,1}$ $b_{0,0}$
$a_{0,2}$ $a_{0,1}$ $a_{0,0}$
$a_{1,2}$ $a_{1,1}$ $a_{1,0}$
$a_{2,2}$ $a_{2,1}$ $a_{2,0}$

Figure 4: 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

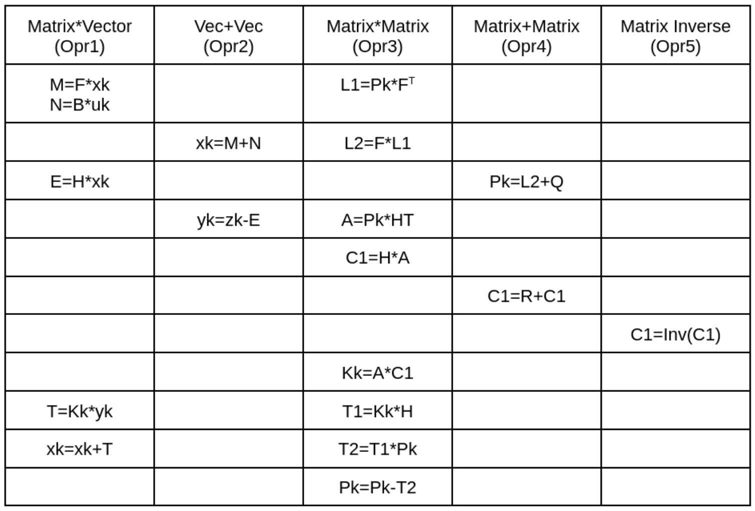
Several architectures were explored (such as QR and LU decomposition) but we ended up going with Gauss Jordan elimination using row operations, as it seemed to be easiest to implement and not a lot of latency was sacrificed. For a non-singular matrix M of dimensions $n \times n$, the algorithm works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

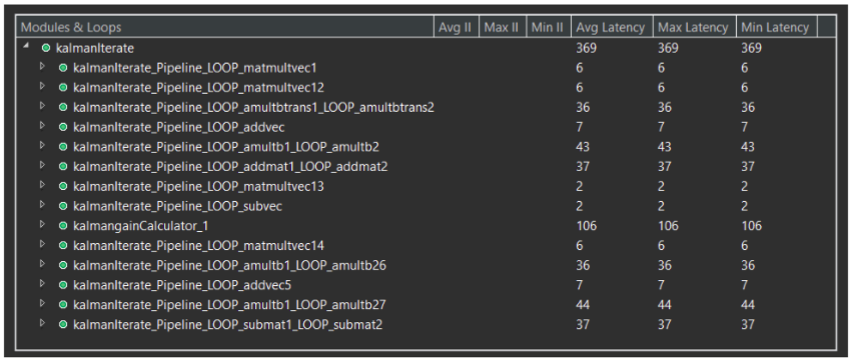
	xk=M+N	L2=F*L1		
E=H*xk			Pk=L2+Q	
	yk=zk-E	A=Pk*HT		
		C1=H*A		
			C1=R+C1	
				C1=Inv(C1)
		Kk=A*C1		
T=Kk*yk		T1=Kk*H		
xk=xk+T		T2=T1*Pk		
		Pk=Pk-T2		

Figure 5: Task schedule for the top module

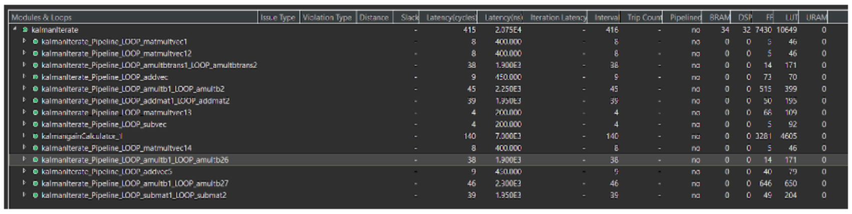
The rules in the top module are based on the functions used in the C code and are scheduled according to the table above.

4 Synthesis using Vitis HLS

Synthesis using Vitis HLS was done based on the baseline C code. We used floating point operations here. For 6 state dimensions, 2 measurement dimesions and 6 input dimensions, Vitis produced the following results:



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: Vitis results

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

The above results were obtained with bare minimum optimizations and passed the test case. Various optimizations like unrolling, pipelining, software pipelining and parallelism were tried through DATAFLOW, PIPELINE, UNROLL, DEPENDENCE pragmas. Any further optimization resulted in dependency errors which we were unable to resolve in the given time.

Part III

Conclusion

One of the real-time applications for the Kalman filter is Simultaneous Localization in mobile robots. An extreme case of such an example would be Autonomous Race Cars where the frequency of operation would be nominally around 200 Hz {5ms} with the state, input, and measurement vector dimensions being around 12-15 variables{x, y, pitch, roll, yaw, and their derivatives}. In such a case, a simple Single-threaded C implementation cannot be implemented because of the high latency associated with them - 0.2 ms based on our calculations{we considered lower-dimensional measurement and input matrices in our profiling} which is comparable. HLS, Bluespec, and Task-Parallelized implementations can be used.

Part IV

Work distribution and links

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): Surya, Saurav
- Analysis: Saurav, AMS
- Vitis HLS: AMS, Surya
- Profiling: AMS