This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCSII.2016.2565730, IEEE Transactions on Circuits and Systems II: Express Briefs

A FPGA-based Unscented Kalman Filter for System-On-Chip Applications

Jeremy Soh, Xiaofeng Wu

***Abstract*—Demand for fast and accurate state estimation in embedded systems has been increasing lately, at least in part due to mobile robotics such as UAVs. The desire to maintain high performance but with compact form factors leads to implementation issues especially with more complex systems. A hardware based approach using Field Programmable Gate Arrays (FPGAs) may be able to alleviate these issues but tends to have a more complicated development process than traditional software-based approaches. In order to simplify development and promote portability between embedded applications, a hardware/software co-design of the Unscented Kalman Filter (UKF) is presented. An example implementation ($N = 18$) of the hardware IP core only is presented using the Zynq-7000 XC7Z045 with synthesis, power and timing results for the 1, 2, 5, and 10 processing element (PE) cases.**

I. Introduction

The demand for fast and accurate state estimation in embedded systems has been increasing due to greater development of autonomous systems such as UAVs and other mobile robotics. These applications, UAVs in particular, tend to desire high computing performance, to enhance capabilities, but still retain light and compact form factors. This often places harsh restrictions on the physical space and electrical power available to the computing system; in particular distributed computing systems. These restrictions mean there may be insufficient computing power available to run complex estimation algorithms, which would be ideal for the application; for example multi-sensor UAVs for autonomous aerial mapping.

Implementing functionality as hardware has the potential to alleviate some of these problems by allowing the use of these complex algorithms at a reasonable speed or by allowing multiple functions to be implemented on a single chip, in a System-on-Chip (SoC). While the hardware approach often offers performance gains over the software approach in terms of speed and power, and in some cases physical space, it also tends to greatly increase development time/complexity and limits reusability. FPGAs certainly reduces the impact of these disadvantages compared to the traditional Application Specific Integrated Circuit (ASIC) approach, but even FPGAs still have long and complex development times compared to software approaches. A mixed software/hardware design may be able to utilise the advantages of both approaches, in terms of speed, power and development effort, as well as promote portability between applications.

A popular estimation method for many applications has been the Kalman Filter family of algorithms, particularly the Extended Kalman Filter (EKF) and, more recently, the Unscented Kalman Filter (UKF) variants. Both the EKF and UKF variants handle nonlinear system models though the UKF has been seen to have superior performance than the EKF, usually at an increased computational cost [1], [2]; a problem for software approaches but hardware approaches may be able to bring computation time back down to reasonable levels.

Previous work by the authors [3] explored the feasibility of a software/hardware co-design of the Unscented Kalman Filter specifically targeted at attitude determination for nanosatellite applications.

This letter presents an extension of the previous work to a more generic design, suitable for many other embedded systems and applications. The presented co-design explores a parallelised (where, due to resource constraints, the previous work had none) datapath and is parameterisable to give a system designer greater flexibility in implementation. The co-design also aims to reduce development complexity and provide greater portability between applications where the UKF may be useful; in addition to space applications as part of a full SoC [3], [4], other examples include speed-ups for vision-based navigation where the UKF would be otherwise infeasible [5] or parallel UKFs in UKF-based particle filters used for SLAM [6]. To emphasise this portability, this letter mainly focuses on the generic hardware part of the implementation.

II. Unscented Kalman Filter

Consider the nonlinear system:

$$\mathbf{x}_k = f\left(\mathbf{x}_{k-1}, \mathbf{w}_{k-1}\right) \tag{1a}$$

$$\mathbf{z}_k = h\left(\mathbf{x}_k, \mathbf{v}_k\right) \tag{1b}$$

where f and h are the process and observation models respectively, $\mathbf{x}_k$ and $\mathbf{z}_k$ are the state and observation vector respectively and $\mathbf{w}_k$ and $\mathbf{v}_k$ are assumed to be zero-mean Gaussian white noise terms.

The EKF handles nonlinear models via linearisation of the system models by Jacobians; this can lead to errors being introduced in the linearisation process. The UKF instead models the current state as a probability distribution characterised by a mean and covariance. A set of points, called sigma points, are deterministically drawn from this distribution and propagated through the system models; the mean and covariance of the transformed points inform the new state. Some expressions that are referred to later in this letter are reproduced for convenience but for interested readers, further details on the UKF can be found in [7], [8].

The sigma points are generated via:

$$\boldsymbol{\mathcal{X}}_{i,k} = \hat{\mathbf{x}}_k^a + \left(\sqrt{\mathbf{P}_k^a}\boldsymbol{\sigma}\right)_i \qquad i = 0, \ \ldots, \ N+1 \tag{2}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCSII.2016.2565730, IEEE Transactions on Circuits and Systems II: Express Briefs

where $\hat{\mathbf{x}}_k^a$ is the augmented state vector (including the state and noise terms), $\mathbf{P}_k^a$ is the augmented covariance, N is the length of the augmented state vector and i refers to the i-th column of the matrix product. The matrix σ is a weighting matrix dependant on the sigma points selection strategy used. Here we use the spherical simplex set of points [9] which utilises a minimal set of sigma points ($N+2$).

The mean and covariance of the transformed sigma points, after propagation through system models, f and h, are recovered by:

$$\hat{\mathbf{x}}_k = \sum_{i=0}^{N+1} W_i \boldsymbol{\mathcal{X}}_{i,k|k-1} \tag{3}$$

$$\mathbf{P}_k = \sum_{i=0}^{N+1} W_i \left[\boldsymbol{\mathcal{X}}_{i,k|k-1} - \hat{\mathbf{x}}_k\right]\left[\boldsymbol{\mathcal{X}}_{i,k|k-1} - \hat{\mathbf{x}}_k\right]^T \tag{4}$$

where W is another weighting coefficient dependant on the sigma points selection strategy.

III. Design

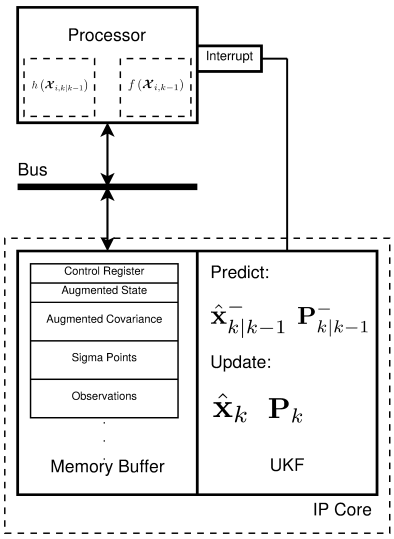

Fig. 1. Block diagram of the top level implementation. The processor implements the propagation of sigma points through the system models, f and h, as software. All other calculations are implemented as hardware by the IP core.

The design is split into two parts: a software part on some processor and a hardware part that assists the processor. The software part contains an implementation of the application-specific system models as software . The hardware part contains the rest of the calculations implemented as an IP core and attached to the processor, as a slave, over some communication bus (see Figure 1).

The core contains a memory buffer to allow the processor to send or retrieve data, as well as control information, to or from the core. The memory buffer has an internal memory map to ensure the control information and data is coherent between the processor and the IP core (see Figure 1). The control register allows the processor to reset or enable the IP core as a whole as well as start one of the core's functional steps via a state machine. The control register also records the current state of the IP core. Data required by the IP core (e.g. transformed sigma points) must be placed in the memory buffer at the appropriate address by the processor before signalling the IP core to begin its calculations.

The control register may be polled by the processor to control the IP core; alternatively, the core may also be configured with an optional interrupt line that may be attached to the processor's interrupt controller or external interrupt lines. The core is further parameterisable to use multiple processing elements (PE), within the datapath, at the designer's discretion.

A configuration file (e.g. plain-text) is used to record relevant parameters for the application (e.g. number of state variables, sigma points weights, memory offsets etc.). A configuration script is then used generate header files for both the hardware and software parts, ensuring parameters are identical between the two parts. The script may also be used to export initialisation data for the software part.

A. State machine

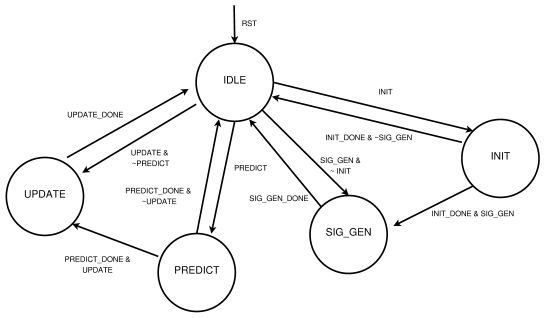

Fig. 2. State machine controlling the IP core

The IP core is controlled by a state machine (see Figure 2) that waits for instructions from the processor before beginning one of the UKF steps: `init`, `sig_gen`, `predict`, or `update`. All transitions to start a new step require the relevant bit to be set in the control register, while all 'done' transitions are subsequently recorded in the control register as well.

During the `init` state the processor may initialise the internal memory of the IP core with initial values for the augmented state and covariance. The `sig_gen` state handles the calculation of the latest set of sigma points (see (2)). The `predict` state uses the newly generated sigma points to calculate the apriori state and covariance ((7) - (9) from [3]). Similarly, the `update` state uses the sigma points to calculate the current state and covariance ((10) - (17) from [3]).

Assuming valid data has been placed in the buffer, the `init` state may be performed in conjunction with the `sig_gen` step to either start new or reset old state estimates, otherwise the `sig_gen` step utilises previously calculated values for the augmented state and covariance. Similarly, the `predict` and `update` steps may be performed together if valid observations are available or independently as required.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCSII.2016.2565730, IEEE Transactions on Circuits and Systems II: Express Briefs

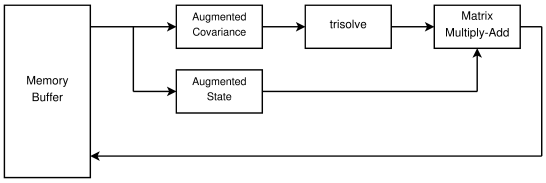

Fig. 3. Block diagram of the `sig_gen` step

B. Sigma points generation

The generation of sigma points involves taking the matrix 'square-root' of the augmented covariance then multiplying the result by a weighting matrix before adding the augmented state column-wise. If the core's internal storage is initialised first, previously stored estimates are overwritten before continuing. After the sigma points are calculated, they are written to the internal storage as well as the memory buffer, a control bit set to signify completion to the processor and, if the interrupt line is included, an interrupt generated. A block diagram of this step can be seen in Figure 3.

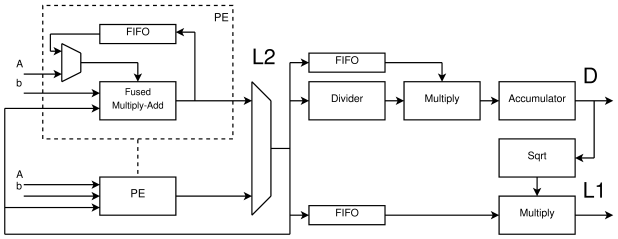

Fig. 4. Datapath for the `trisolve` operation. For the Cholesky Decomposition, the first set of processing elements (PE) calculate the elements of L_2 while the second part calculates the diagonals of D then recombines the two results to calculate L_1

1) Triangular linear equations solver: The calculation of the matrix 'square-root' for positive definite matrices is usually performed via Cholesky Decomposition [10]. In addition to the matrix 'square-root', the Choleksy Decomposition is also used in the Kalman gain calculation which involves a matrix inversion (see [8]). Here, rather than directly inverting the matrix, an algorithm called the matrix 'right divide' is used. For positive definite matrices this algorithm involves using the Cholesky Decomposition to decompose the target matrix (i.e. the matrix to be inverted) into triangular form followed by forward elimination then back substitution to solve the system.

The Cholesky Decomposition, however, contains a division and a (scalar) square-root operation in its datapath which limits performance since each row of the decomposition must be calculated sequentially, i.e. rows cannot be calculated in parallel. So instead of the original decomposition:

$$A = L_1 L_1^T \tag{5}$$

where L_1 is lower triangular, an alternative version of the decomposition is used:

$$A = L_2 D L_2^T \tag{6}$$

where L_2 is lower triangular and its diagonal terms are unit elements, D is diagonal and the two versions are related by $L_1 = L_2\sqrt{D}$. This allows the Cholesky Decomposition, forward elimination and back substitution operations to all be treated as solving a series of triangular linear equations [11]; i.e. the equation:

$$L_2 D L_2^T x = b \tag{7}$$

is equivalent to solving:

$$L_2 c = b \tag{8a}$$

$$De = c \tag{8b}$$

$$L_2^T x = e \tag{8c}$$

Perhaps more importantly, this means the division and (scalar) square-root operations can be moved out of the main datapath and calculated separately; Figure 4 depicts the full `trisolve` datapath including the divsion and (scalar) square-root operations. Despite the reshuffle of operations, the Cholesky Decomposition still has the most potential to limit performance of the core due to, as mentioned, the inability to parallelise row calculations.

In the `trisolve` datapath, the first set of PEs calculate the whole of L_2, in the case of the decomposition, or c, in the case of forward elimination; the number of PEs is up to the designer and is controlled via a parameter. The second half of the datapath calculates the diagonals of D, in the case of the decomposition, or e; the first half can be reused to calculate x finally. The second half also recombines L_2, D to recover the original Cholesky Decomposition products where necessary.

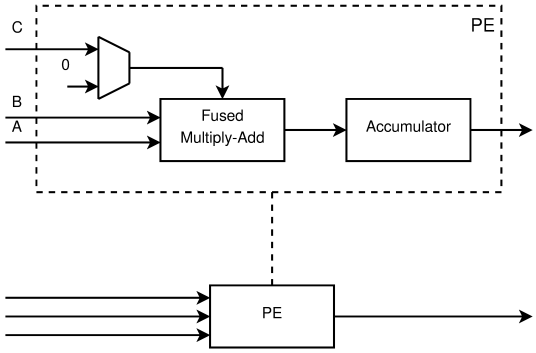

Fig. 5. Datapath for the matrix multiply-add operation

2) Matrix multiply-add: The matrix multiply-add datapath is a standard element-wise multiplication and accumulation (see Figure 5). The processing elements simply inject elements of the matrix to be added into the accumulation directly instead of performing an additional matrix addition after a matrix multiplication. Each processing element calculates one row of the result matrix at a time; i.e. multiple processing elements calculate multiple rows in parallel.

C. Predict

The architecture for the `predict` step can be seen in Figure 6. The processor may initiate a `predict` step once it has read the sigma points generated from the previous step, propagated them through the `predict` model, f, and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCSII.2016.2565730, IEEE Transactions on Circuits and Systems II: Express Briefs

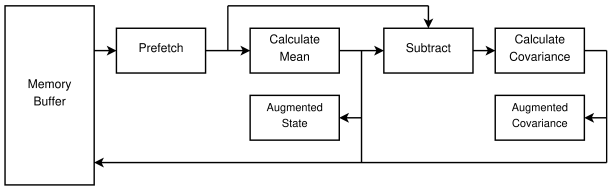

Fig. 6. Block diagram of the `predict` step

has written the transformed points into the memory buffer. The prefetch module fetches the transformed sigma points from the memory buffer and places them into a parallel memory structure. The mean of the transformed sigma points is calculated and written back to the augmented state vector memory and the memory buffer as the apriori state estimate. The mean and transformed sigma points are used to calculate the residuals (e.g. $\boldsymbol{\mathcal{X}}_{i,k|k-1} - \hat{\mathbf{x}}_k$, see (4)), using a simple `subtract` operation, before the covariance calculation. The new covariance is written back to the augmented covariance memory and the memory buffer as the apriori covariance; in this way both the internal memory and the processor has the most recent state estimate and covariance. As with the previous step, once the `predict` step is completed a control bit is set to notify the processor and, if included, an interrupt generated.

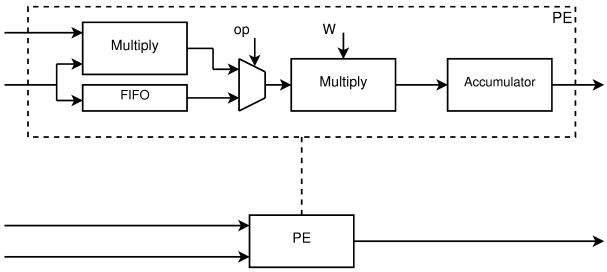

Fig. 7. Datapath for calculation of the mean and covariances. `op` controls which result is output, W refers the sigma points weights W_0, W_1 which are stored as constants

1) Calculation of mean/covariance: The mean and covariance calculations (e.g. (3) and (4)) simply involve a series of multiplications and accumulation once the residuals have been calculated which is why the residuals are calculated first; the datapath used for these calculations can be seen in Figure 7. The input is either the transformed sigma points to calculate the mean, or the residuals to calculate the covariance. A parameter controls the number of PEs in parallel.

D. Update

The architecture for the `update` step can be seen in Figure 8. The processor may begin an `update` step once it has written sigma points transformed by the observation model into the memory buffer. Similar to the `predict` step, a prefetch stage converts the data into a parallel memory structure. The mean and residuals are calculated as before, then used to calculate the observation covariance. These 'observation' residuals are combined with the 'state' residuals, which were calculated during the `predict` step, to calculate the cross covariance between the two system models. These two covariances are used to calculate the Kalman gain before the final matrix multiply-add stages use the Kalman gain and the apriori state estimate and covariance to calculate the new state estimate and covariance. The new estimates overwrite the apriori estimates in the internal memory and are also written into the memory buffer such that both the core and the processor have the most recent estimate. The core notifies the processor upon completion in the same way as before, setting a control bit and/or generating an interrupt.

IV. Example Implementation

An example implementation is presented here using the Xilinx Zynq-7000 series XC7Z045 [12]; in this example details of only the hardware IP core are presented since the software part is highly application dependant. To set some of the relevant parameters, consider for example, an application using 3-axis attitude and angular velocity (6 state variables), two sets of sensor measurements for both (12 observation variables) and ideal system models f and h, i.e. no errors; the length of the augmented state vector, N, is then 18. The core uses a single precision floating point representation and is attached to the processor using the AXI4 protocol. The core was developed in Verilog and synthesised using Xilinx's Vivado 2014.1; basic arithmetic (i.e. single precision add, multiply etc.) is implemented using Xilinx's IP catalogue. The implementation strategy used was to minimise logic area and configuration scripts in Matlab were used to generate headers for both the core and the, here unused, software part.

A. Synthesis

TABLE I
Synthesis Results

Resource	1 PE	2 PE	5 PE	10 PE
FF	8307 (2%)	14392 (3%)	27502 (6%)	49061 (11%)
LUT	6299 (3%)	14824 (7%)	29026 (13%)	52554 (24%)
BRAM	16.5 (3%)	39 (7%)	67.5 (12%)	120 (22%)
DSP48	35 (4%)	66 (7%)	108 (12%)	184 (20%)

Synthesis results for a range of processing elements is given in Table I; for simplicity, each module has the same number of processing elements. The results do not include a processor but do include the logic used for the AXI4 interface. As noted earlier, the implementation strategy was to minimise logic (LUTs, FFs) but still resulted in a large amount of internal memory usage, especially as more PEs are used. Additionally, a small amount of logic overhead is introduced when using multiple PE's but this overhead is much less significant as more PE's are used.

B. Power

A power estimate and breakdown using the Xilinx Power Analyzer can be seen in Table II. The power breakdown only includes the dynamic power use by the design, i.e. it excludes power usage by the processor as well as device static power dissipation. A small amount of overhead is introduced when using multiple PE's, otherwise the power increases proportionally as the number of PE's increase.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCSII.2016.2565730, IEEE Transactions on Circuits and Systems II: Express Briefs

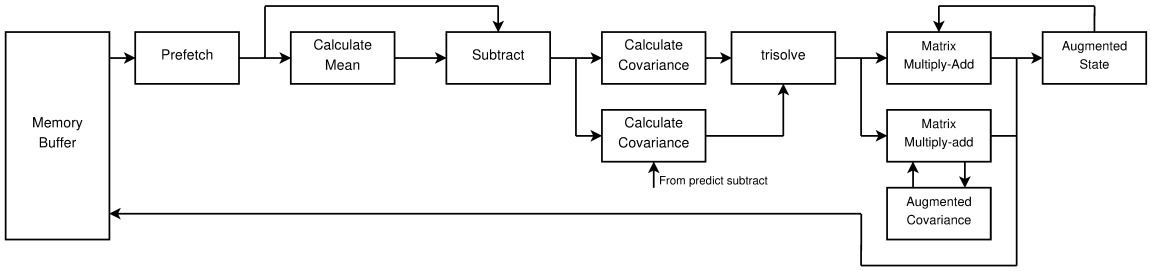

Fig. 8. Block diagram of the `update` step

TABLE II
POWER CONSUMPTION

	1 PE (mW)	2 PE (mW)	5 PE (mW)	10 PE (mW)
Clocks	44	74	134	233
Signals	29	83	152	249
Logic	28	75	128	208
BRAM	25	101	180	338
DSP	5	22	34	56
Total	131	355	628	1084

C. Timing

TABLE III
TIMING PERFORMANCE. ALL VALUES IN μS.

	SW	1 PE	2 PE	5 PE	10 PE
Sigma Points Generation	402	116	72	52	43
Predict	31	15	9	8	5
Update	174	115	76	52	44
Total	606	246	157	112	92

An estimate of the latency across each of three steps can be seen in Table III. The maximum synthesisable frequency is 100 MHz and the latency is measured via simulation using arbitrary data. Multiple PE's initially provide an appropriate speed-up but rapidly suffer diminishing returns as the number of PE's increases. The core spends the majority of its time in the `sig_gen` and `update` steps predominantly because of the `trisolve` module.

For comparison, a purely software (C) implementation of the non-application specific calculations (i.e. ignoring system models f and h as before) is run on the processor system featured by the XC7Z045 (an ARM Cortex-A9 @ 667 MHz). It can be seen that the hardware IP core is a little over twice as fast even for the single PE case despite the slower clock speed.

As noted earlier, the Cholesky Decomposition is calculated one row at a time so increasing the number of PE's does little to speed-up the module further; this is predominantly the cause of the diminishing returns of the timing gains. Increasing the number of PE's does, however, still increase the performance of the `trisolve` module when performing the forward elimination or back substitution. The largest timing gains occur when the number of PE's are equal to (or greater than) the number of state variables or the number of observation variables, whichever is larger. It may also be better to instantiate a different number of PE's for different modules as it may be possible in some cases to preserve timing gains but save on logic area further.

V. CONCLUSION

In this letter, a generic software/hardware co-design of the Unscented Kalman Filter was presented. The design aims to combine portability and quick development times, typical benefits of full software designs, with high performance, a typical benefit of full hardware designs. An example implementation of the hardware IP core only was presented; increasing the number of processing elements used increases the logic area and power consumption proportionally, however the timing performance suffers diminishing returns due to the nature of the Cholesky Decomposition calculation.

REFERENCES

[1] J. Crassidis, F. Markley, and Y. Cheng, "Survey of nonlinear attitude estimation methods," *Journal of Guidance Control and Dynamics*, vol. 30, no. 1, p. 12, 2007.

[2] R. Kandepu, B. Foss, and L. Imsland, "Applying the unscented kalman filter for nonlinear state estimation," *Journal of Process Control*, vol. 18, no. 78, pp. 753 – 768, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0959152407001655

[3] J. Soh and X. Wu, "A Modular FPGA-based Implementation of the Unscented Kalman Filter," in *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*, July 2014, pp. 127–134.

[4] ——, "A FPGA-based approach to attitude determination for nanosatellites," in *Industrial Electronics and Applications (ICIEA), 2012 7th IEEE Conference on*, 2012, pp. 1700–1704.

[5] S. Holmes, G. Klein, and D. Murray, "An $O(N^2)$ Square Root Unscented Kalman Filter for Visual Simultaneous Localization and Mapping," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 31, no. 7, pp. 1251 –1263, july 2009.

[6] C. Kim, R. Sakthivel, and W. K. Chung, "Unscented FastSLAM: A Robust and Efficient Solution to the SLAM Problem," *Robotics, IEEE Transactions on*, vol. 24, no. 4, pp. 808–820, Aug 2008.

[7] E. Wan and R. Van Der Merwe, "The unscented kalman filter for nonlinear estimation," in *Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*. IEEE, 2000, pp. 153–158.

[8] S. Julier and J. Uhlmann, "Unscented filtering and nonlinear estimation," *Proceedings of the IEEE*, vol. 92, no. 3, pp. 401–422, 2004.

[9] S. Julier, "The spherical simplex unscented transformation," in *American Control Conference, 2003. Proceedings of the 2003*, vol. 3, June 2003, pp. 2430–2434 vol.3.

[10] G. H. Golub and C. F. Van Loan, *Matrix computations*, 3rd ed. Baltimore: Johns Hopkins University Press, 1996.

[11] D. Yang, G. Peterson, H. Li, and J. Sun, "An FPGA Implementation for Solving Least Square Problem," in *Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on*, april 2009, pp. 303 –306.

[12] *Zynq-7000 All Programmable SoC Technical Reference Manual*, Xilinx, September 2013, UG585 (v1.6.1).