We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,900
Open access books available

185,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities



WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com



Chapter

A Scalable, FPGA-Based Implementation of the Unscented Kalman Filter

Jeremy Soh and Xiaofeng Wu

Abstract

Autonomous aerospace systems may well soon become ubiquitous pending an increase in autonomous capability. Greater autonomous capability means there is a need for high-performance state estimation. However, the desire to reduce costs through simplified development processes and compact form factors can limit performance. A hardware-based approach, such as using a field-programmable gate array (FPGA), is common when high performance is required, but hardware approaches tend to have a more complicated development process when compared to traditional software approaches; greater development complexity, in turn, results in higher costs. Leveraging the advantages of both hardware-based and software-based approaches, a hardware/software (HW/SW) codesign of the unscented Kalman filter (UKF), based on an FPGA, is presented. The UKF is split into an application-specific part, implemented in software to simplify the development process, and a non-application-specific part, implemented in hardware as a parameterisable 'black box' module (i.e. IP core) to increase performance. Simulation results demonstrating a possible nanosatellite application of the design are presented; implementation (synthesis, timing, power) details are also presented.

Keywords: field-programmable gate array (FPGA), unscented Kalman filter (UKF), codesign, system on a chip (SoC), nonlinear state estimation

1. Introduction

Small (micro-, nano-, pico-) satellites and (micro-) unmanned aerial systems (UASs) are emerging technologies that have the potential to be of great academic and commercial use but only if a balance can be found between two diametrically opposed forces that act on their design: the desire, and need, for high performance and the desire to reduce costs. High performance, especially in state estimation, is necessary for these technologies to be advantageous over traditional aerospace systems in relevant applications.

The desire to reduce the costs of these technologies has led to their miniaturisation and heavy use of commercial off-the-shelf (COTS) components so that some level of economy of scale may be achieved. Though both component and development costs can be reduced in this way, this approach, in turn, leads to a reduction in the resources available (e.g. electrical power, computing power and physical space) aboard those systems, impacting performance.

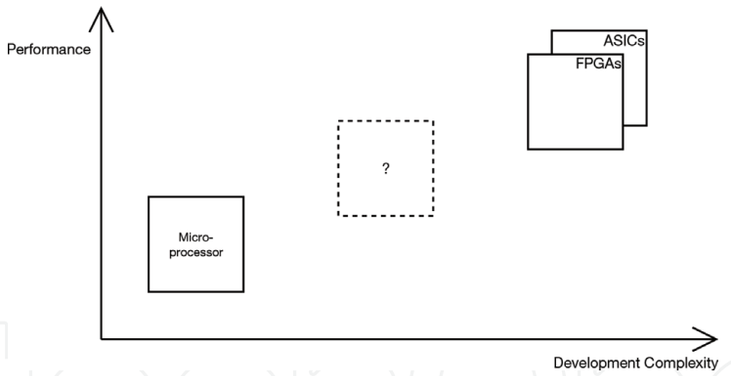


Figure 1.
The performance versus development complexity trade-off for different types of embedded systems.

Specialised hardware, e.g. application-specific integrated circuit (ASIC)- or field-programmable gate array (FPGA)-based systems, can achieve high performance, even for severely resource-constrained systems, but tends to increase the development complexity of these systems; in this way, using specialised hardware may reduce component costs and meet performance and miniaturisation requirements, while development costs are typically increased. This issue is illustrated in **Figure 1**, which depicts the balance between development complexity and performance for different embedded systems; greater complexity during the development process means that a greater investment in resources, personnel and time becomes necessary, which leads to higher development costs.

Software approaches, e.g. microprocessor-based systems, generally have lower performance than specialised hardware but have much simpler, and thus cheaper, development processes. It is, however, possible to draw upon aspects of both hardware and software approaches and combine them into a hardware/software codesign. This codesign could deliver the high performance of specialised hardware but, by using software techniques, e.g. modularity or abstraction, could also alleviate some of the high development costs associated with such hardware. If this codesign approach is applied to a prolific state estimation algorithm, then the performance and miniaturisation requirements could be met, while keeping development costs low.

In this chapter, a library containing a scalable, hardware/software (HW/SW) codesign of the unscented Kalman filter (UKF), based on an FPGA, is presented. The codesign is implemented as a fully parameterisable, self-contained 'black box' (which is often referred to as an IP core), which aims to minimise the necessary input from system designers when applying the codesign to a new application, such that overall development complexity is reduced.

This chapter is a distillation of work first presented in [1]. The rest of this chapter is organised as follows: Section 2 provides background on relevant concepts, Section 3 outlines the proposed design, Section 4 describes the simulation model to verify the design and simulation results, Section 5 presents implementation results and Section 6 concludes the chapter.

2. Background

2.1 Hardware/software codesign

Hardware/software codesign is a design practice that is often used with system-on-a-chip (SoC) architectures. The term system on a chip comes from the field of very large-scale integration (VLSI) where individual hardware units or 'black

boxes' (IP cores) that perform some dedicated function are arranged and connected together on a single ASIC chip. Typical SoCs may include a microcontroller or microprocessor core, DSPs, memories such as RAMs or ROMs, peripherals such as timers/counters, communication interfaces such as USB/UART, analog interfaces such as ADCs/DACs or other analog, digital, mixed-signal or RF hardware. Previously, each of these components may have had its own ASIC and was connected together on a PCB, but, in accordance with Moore's law, resource densities of silicon chips have massively increased over time, so now these components are able to be integrated together on a single chip; an example SoC can be seen in **Figure 2a**.

(a) Example of a typical system on a chip.
(b) An example of a target architecture for early hardware/software codesign implementations.

SoC designs for FPGAs have become more popular recently as the increase in resource densities allowed more complex logic to be implemented. The push towards SoCs on FPGAs is driven by the desire for greater autonomy in a variety of systems; the most obvious example is field robotics, but autonomous systems such as 'smart' rooms and satellites also have a need for a small form factor and high-performance computing solutions that the FPGA is well placed to deliver.

As VLSI technology matured, designers began to see that the increase in development complexity for hardware or ASIC designs was impacting their ability to bring products to market quickly. The associated increase in the complexity of microprocessors led many designers to realise that these microprocessor units could be included into system designs and some of the functionality shifted to software to reduce their time to market. Microprocessors can be considered a SoC on their own, but they can also be included in much larger SoC designs, and this is where the idea of hardware/software codesign first began; reviews of the field by [2–4] give a comprehensive history of hardware/software codesign. A basic example of an architecture where hardware/software codesign may be appropriate is shown in **Figure 2b**.

2.2 Unscented Kalman filter

The extended Kalman filter (EKF) is, and has been, the most widespread method for nonlinear state estimation [5]. It has also become the de facto standard

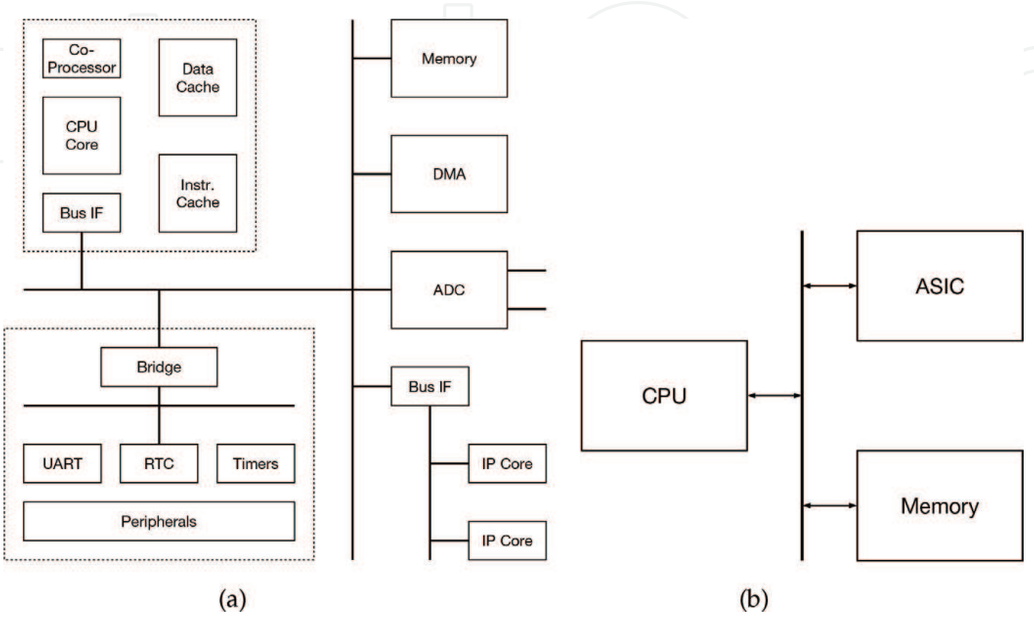


Figure 2.
Examples of system-on-a-chip architectures.

by which other methods are compared when analysing their performance. Various surveys of the field have noted that the EKF is 'unquestionably dominant' [6], 'the workhorse' of state estimation [7, 8] and the 'most common' nonlinear filter [9]. Despite some shortcomings, the relative ease of implementation and still-remarkable accuracy have propelled the EKF's popularity.

However, more recently the unscented Kalman filter (UKF) [10] has been shown to perform much better than the EKF when the system models are 'highly' nonlinear [6–9, 11]. While the EKF attempts to deal with non-linearities in the system model by using the Jacobian to linearise the system model, the UKF instead models the current state as a probability distribution with some mean and covariance. Following this, a deterministic sampling technique known as the unscented transform (UT) is applied. A set of points, called 'sigma' points, are drawn from the probability distribution, and each of them propagated through the nonlinear system models. The new mean and covariance of the transformed sigma points are then recovered to inform the new state estimate. The crucial aspect of the UT is that the sigma points are drawn deterministically, unlike random sampling methods like Monte Carlo algorithms, drastically reducing the number of points necessary to recover the 'transformed' mean and covariance.

Consider the general nonlinear system described for discrete time, k:

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

$$\mathbf{z}_k = h(\mathbf{x}_k, \mathbf{v}_k) \tag{2}$$

where f and h are the system's process and observation models, respectively; $\mathbf{x}$ and $\mathbf{z}$ are the state and observation vectors, respectively; $\mathbf{u}$ is the control input and $\mathbf{w}$ and $\mathbf{v}$ are, respectively, the process/control and measurement/observation noise, which are assumed to be zero-mean Gaussian white noise terms with covariances $\mathbf{Q}$ and $\mathbf{R}$. The formalisation of the UKF for this system is as follows. Define an augmented state vector, $\mathbf{x}^a$, with length M that concatenates the process/control noise and measurement noise terms with the state variables as

$$\mathbf{x}_k^a = \begin{bmatrix} \mathbf{x}_k \\ \mathbf{w}_k \\ \mathbf{v}_k \end{bmatrix} \tag{3}$$

The augmented state vector has an associated augmented state covariance, $\mathbf{P}_k^a$, which combines the (regular) state covariance, $\mathbf{P}_k$, with the noise covariances $\mathbf{Q}_k$ and $\mathbf{R}_k$. The augmented state vector and covariance are initialised with

$$\begin{aligned} \hat{\mathbf{x}}_0^a &= E\left[\mathbf{x}_0^a\right] = \begin{bmatrix} \hat{\mathbf{x}}_0 \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix} \\ \mathbf{P}_0^a &= E\left[\left(\mathbf{x}_0^a - \hat{\mathbf{x}}_0^a\right)\left(\mathbf{x}_0^a - \hat{\mathbf{x}}_0^a\right)^T\right] = \begin{bmatrix} \mathbf{P}_0 & 0 & 0 \\ 0 & \mathbf{Q}_0 & 0 \\ 0 & 0 & \mathbf{R}_0 \end{bmatrix} \end{aligned} \tag{4}$$

where $\hat{\mathbf{x}}_0$ is the expected value of the initial (regular) state. There exist various other sigma point selection strategies, and, in order to minimise computational effort, a selection strategy involving a minimal set of samples is highly desired. The spherical simplex set of points [10, 12] can be shown to offer similar performance to

the original UKF with the smallest number of sigma points required $(M + 2)$. The sigma point weights, and a coefficient matrix is generated by choosing $0 \leq W_0 \leq 1$ and then calculating W_1:

$$W_1 = W_i = \frac{(1 - W_0)}{(M + 1)} \qquad i = 1, ..., M + 1 \tag{5}$$

The choice of W_0 determines the spread of sigma points about the mean. Choosing $W_0 \approx 1$ reduces the spread, implying a greater confidence in the previous estimate, while the opposite is true when choosing $W_0 \approx 0$. The vector sequence is initialised as

$$\boldsymbol{\sigma}_0^1 = [0], \quad \boldsymbol{\sigma}_1^1 = -\left[\frac{1}{\sqrt{2W_1}}\right], \quad \boldsymbol{\sigma}_2^1 = \left[\frac{1}{\sqrt{2W_1}}\right] \tag{6}$$

Then, the vector sequence is expanded for $j = 2, ..., M$ via

$$\boldsymbol{\sigma}_i^j = \begin{cases} \begin{bmatrix} \boldsymbol{\sigma}_0^{j-1} \\ 0 \end{bmatrix} & i = 0 \\ \begin{bmatrix} \boldsymbol{\sigma}_i^{j-1} \\ -\dfrac{1}{\sqrt{j(j+1)W_1}} \end{bmatrix} & i = 1, ..., j \\ \begin{bmatrix} \mathbf{0}_{j-1} \\ \dfrac{j}{\sqrt{j(j+1)W_1}} \end{bmatrix} & i = j + 1 \end{cases} \tag{7}$$

The actual sigma points are drawn, via a column-wise accumulation, from

$$\boldsymbol{\chi}_{i,k} = \hat{\mathbf{x}}_k^a + \left(\sqrt{\mathbf{P}_k^a}\boldsymbol{\sigma}\right)_i \tag{8}$$

where i refers to the ith column of the matrix product and $\sqrt{\mathbf{P}_k^a}$ refers to the matrix 'square root'. The matrix square root of a target matrix $\mathbf{A}$ is a matrix $\mathbf{B}$ that satisfies $\mathbf{A} = \mathbf{BB}$; it is often calculated via the Cholesky decomposition [13].

The `predict` step begins with the sigma points being propagated through the system model:

$$\boldsymbol{\chi}_{i,k|k-1}^x = f\left(\boldsymbol{\chi}_{i,k-1|k-1}^x, \mathbf{u}_{k-1|k-1}, \boldsymbol{\chi}_{i,k-1|k-1}^w\right) \tag{9}$$

The state and covariance are then predicted as

$$\hat{\mathbf{x}}_k^- = \sum_{i=0}^{N-1} W_i^{(m)} \boldsymbol{\chi}_{i,k|k-1}^x \tag{10}$$

$$\mathbf{P}_k^- = \sum_{i=0}^{N-1} W_i^{(c)} \left[\boldsymbol{\chi}_{i,k|k-1}^x - \hat{\mathbf{x}}_k^-\right]\left[\boldsymbol{\chi}_{i,k|k-1}^x - \hat{\mathbf{x}}_k^-\right]^T \tag{11}$$

For the `update` step, the sigma points that were updated in the `predict` step are propagated through the observation model:

$$\boldsymbol{\mathcal{Z}}_{i,k|k-1} = h\left(\boldsymbol{\chi}_{i,k|k-1}^x, \boldsymbol{\chi}_{i,k-1|k-1}^v\right) \tag{12}$$

The mean and covariance of the observation-transformed sigma points are calculated:

$$\hat{\mathbf{z}}_{k|k-1} = \sum_{i=0}^{N-1} W_i^{(m)} \mathcal{Z}_{i,k|k-1} \tag{13}$$

$$\mathbf{S}_{k|k-1} = \sum_{i=0}^{N-1} W_i^{(c)} \left[\chi_{i,\,k|k-1} - \hat{\mathbf{z}}_{k|k-1}\right] \left[\mathcal{Z}_{i,k|k-1} - \hat{\mathbf{z}}_{k|k-1}\right]^T \tag{14}$$

followed by the cross-covariance

$$\mathbf{P}_{xz,\,k|k-1} = \sum_{i=0}^{N-1} W_i^{(c)} \left[\chi^x_{i,\,k|k-1} - \hat{\mathbf{x}}^-_{k|k-1}\right] \left[\mathcal{Z}_{i,k|k-1} - \hat{\mathbf{z}}_{k|k-1}\right]^T \tag{15}$$

and the Kalman gain

$$\mathbf{K} = \mathbf{P}_{xz,\,k|k-1}\mathbf{S}^{-1}_{k|k-1} \tag{16}$$

Finally, the current system state is estimated by

$$\hat{\mathbf{x}}_k = \hat{\mathbf{x}}^-_{k|k-1} + \mathbf{K}\left(\tilde{\mathbf{z}}_k - \hat{\mathbf{z}}_{k|k-1}\right) \tag{17}$$

where $\tilde{\mathbf{z}}$ is the current set of observations and the current covariance is updated with

$$\mathbf{P}_k = \mathbf{P}^-_{k|k-1} - \mathbf{K}\mathbf{S}_{k|k-1}\mathbf{K}^T \tag{18}$$

$$= \mathbf{P}^-_{k|k-1} - \mathbf{P}_{xz,\,k|k-1}\mathbf{K}^T \tag{19}$$

where the expression for the Kalman gain, Eq. (16), is substituted.

3. Hardware/software codesign of the unscented Kalman filter

The first exercise in the hardware/software codesign is to divide the UKF algorithm into two parts. For maximum performance, it is desirable for as much of the algorithm as possible to be implemented in hardware. However, to maintain portability, any part of the algorithm that is application-specific would be better implemented in software. This is so that the application-specific parts can make use of the faster and simpler development processes that using software entails. Reviewing the UKF algorithm, only the two system models, the `predict` and `update` models, are application-specific.

Apart from the two system models, the rest of the UKF can be viewed as, essentially, a series of matrix manipulations. The only changes to the rest of the UKF when either of the system models changes are the size of the vectors and matrices used in the UKF calculations. The sizes of these vectors and matrices are fixed for a particular formulation of the UKF, and so they can be treated as parameters that are set at synthesis. Fixing the parameters at synthesis means that only the bare minimum of hardware resources is needed, but the hardware can still be easily used for different applications with different vector/matrix sizes; rather than needing to redesign any functionality, the hardware can simply be synthesised with different parameters. Thus, the rest of the UKF can be designed and then used as a parameterisable, modular 'black box' (IP core), and implementing it for any given application only requires the appropriate selection of parameters.

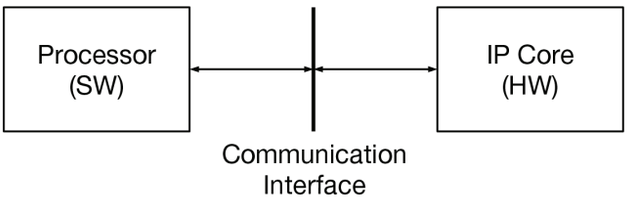


Figure 3.
The hardware/software partition on the FPGA.

When it comes to designing hardware, there are three main considerations: the performance of the hardware (which may include data throughput, maximum clock frequency, etc.), the logic area (on-chip resources) used and the power consumption of the hardware. During development these considerations are usually at odds with each other—specifically performance is usually opposed by logic area and power consumption. In order to increase the performance of the design, additional resources often have to be used which, in turn, may increase the power consumption; these increases in resource/power cost may make the implementation infeasible for a given application. Due to these considerations, it is beneficial to give system designers the ability to scale resource use to the requirements of their particular application.

The actual physical implementation of the hardware/software UKF on an FPGA can be seen in **Figure 3**. The hardware part is implemented as a stand-alone IP core, and the software part is implemented on a general-purpose microprocessor. The processor acts as the main controller which, in addition to implementing the system model software, controls the hardware IP core. The precise method of controlling the IP core is dependent on the design variation and is elaborated on in the following sections.

The processor communicates with the IP core over some communication interface. Any intra-chip communication method would be sufficient and would be driven mostly by the requirements of the application; viable interfaces include point-to-point, bus or NoC interfaces. The IP core contains memory buffers at the interface in order to receive data from the processor as well as to temporarily store data that needs to be read by the processor. The communication interface is the same between all three variants, but the specifics of the memory buffers are not.

Here, the communication interface between the two parts is an AXI4 bus. All variants are implemented using single-precision arithmetic (IEEE-754); this gives a decent balance of dynamic range and resource usage which should be sufficient for the majority of applications. All hardware in the codesign is developed using the Verilog HDL, and all software in the codesign is developed using C. Although C is used here, in general, any type of software may be used as long as it contains the ability to interact with the communication interface connecting the hardware and software parts.

3.1 Overall design

The codesign utilises the main benefit of hardware implementations: wide parallelism. An increase in performance is gained by encapsulating certain parts of the major datapaths into a sub-module called a processing element (PE) and then using multiple instances of these PEs in parallel, allowing multiple elements of an algorithm to be calculated at once. The increase in resources used is not only for the extra processing elements but also in the additional overhead needed to deal with the parallel memory structure that is also necessary to feed to the parallel

processing elements. The number of PEs used in the design is parameterisable, allowing for some trade-offs by the system designer between resources used and performance.

The codesign logically separates the UKF algorithm into three parts, while on the hardware side, the IP core consists of the UKF hardware and a memory buffer which is attached to a communication (AXI4) bus; the top-level block diagram of the codesign can be seen in **Figure 4**. The memory buffer has a memory map that ensures that data are coherent between the processor and the IP core and also incorporate a control register which allows the IP core to be controlled. 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.

3.2 Sigma point generation

The `sig_gen` step uses the current augmented state vector and covariance to calculate the new sigma points via (Eq. (8)). To calculate the new set of sigma points, first the matrix 'square root' of the current augmented covariance must be calculated which is implemented by the `trisolve` module. The 'square root' of the augmented covariance is then multiplied by the sigma coefficients weighting matrix and the current augmented state vector added column-wise; this is implemented by the matrix multiply-add module described in Section 3.2.2. After the sigma points are calculated, they are written to the memory buffer; a control bit is 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 5** showing the data flow between modules.

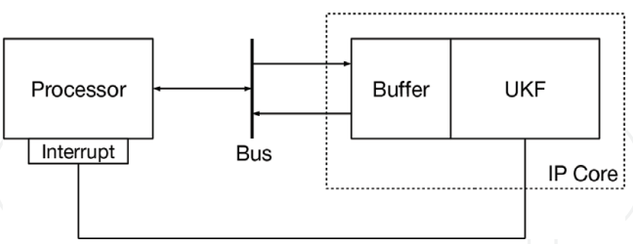


Figure 4.
Top-level block diagram.

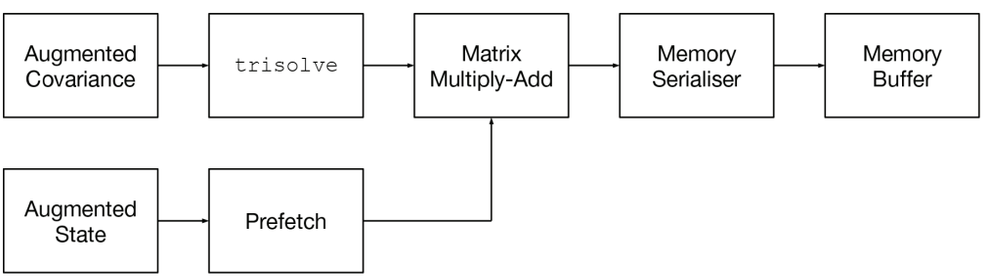


Figure 5.
Block diagram of the `sig_gen` *step.*

The memory prefetch and memory serialiser modules add a small amount of overhead to the `sig_gen` step but are necessary due to the matrix multiply-add featuring a parallelised datapath but the memory buffer requiring serial access.

3.2.1 Triangular linear equation solver

In addition to the matrix 'square root', the Cholesky decomposition is also used in the Kalman gain calculation which involves a matrix inversion (see Eq. (16)). Directly computing a matrix inversion is extremely computationally demanding; so rather than directly inverting the matrix, an algorithm called the matrix 'right divide' is used here. For positive definite matrices, this algorithm involves using the Cholesky decomposition to decompose the target matrix into a triangular form followed by forward elimination and then back substitution to solve the system; this sequence may be treated as solving a series of triangular linear equations meaning the same hardware can be reused for each operation [14]. The Cholesky decomposition of a target matrix $\mathbf{A}$, which is positive definite, is given by

$$\mathbf{A} = \mathbf{L}_1 \mathbf{L}_1^T \tag{20}$$

where $\mathbf{L}_1$ is lower triangular. Reducing the calculation to a series of triangular linear equations involves using an alternative version:

$$\mathbf{A} = \mathbf{L}_2 \mathbf{D} \mathbf{L}_2^T \tag{21}$$

where $\mathbf{L}_2$ is lower triangular and its diagonal terms are unit elements, $\mathbf{D}$ is diagonal and the two versions are related by

$$\mathbf{L}_1 = \mathbf{L}_2 \sqrt{\mathbf{D}} \tag{22}$$

The recombination process is necessary because of the subsequent matrix multiply-add between the augmented covariance square root and the sigma weighting coefficient matrix (see Eq. (8)). The element-wise calculation for $\mathbf{L}_2$ and $\mathbf{D}$ is given by

$$F_{ij} = \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} F_{jk} \right) \quad \text{for } i>j \tag{23}$$

$$D_j = A_{jj} - \sum_{k=1}^{j-1} \frac{F_{jk}^2}{D_k} \tag{24}$$

where $F_{ij} = L_{ij} D_j$ and $F_{jk} = L_{jk} D_k$ are substituted to simplify the calculation further. **Figure 6** depicts the full `trisolve` datapath, including the division and the recombination process to recover $\mathbf{L}_1$. The input b is either A_{ij} in the Cholesky decomposition or an element from the divisor matrix in the matrix 'right divide'.

The fused multiply-add module and feedback FIFO have been encapsulated to form an elementary block of hardware called a processing element (PE) which can be instantiated multiple times in parallel. The PE output to a demultiplexer, which ensures values, is passed to the subsequent calculations in the correct order. The latter calculations, after the demultiplexer, are not parallelised because these calculations require much fewer operations, and so parallelisation is not necessary.

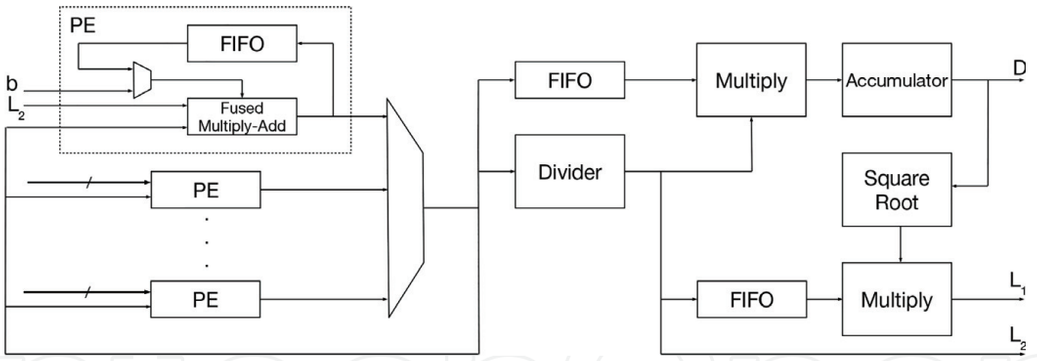


Figure 6.
Triangular linear equation solver.

3.2.2 Matrix multiply-add

The matrix multiply-add datapath is a standard 'naive' element-wise multiplication and accumulation. However, the hardware to calculate one element is enclosed as one processing element, and additional PEs are added to handle calculations in parallel; the matrix multiply-add datapath can be seen in **Figure 7**. Each PE is responsible for calculating at least one row of the result matrix. The elements of the matrix to be added can simply be injected into the accumulation directly, instead of performing an additional matrix addition after a matrix multiplication.

3.3 Predict step

The `predict` step uses the transformed sigma points to calculate the a priori state estimate; the architecture for the `predict` step can be seen in **Figure 8** showing how data flows between each module.

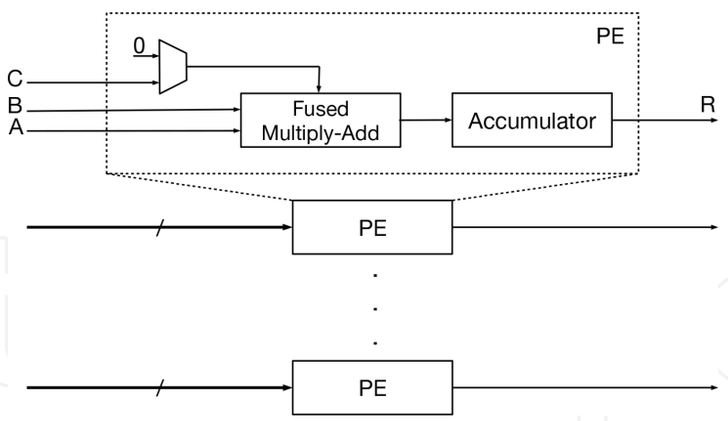


Figure 7.
Matrix multiply-add operation.

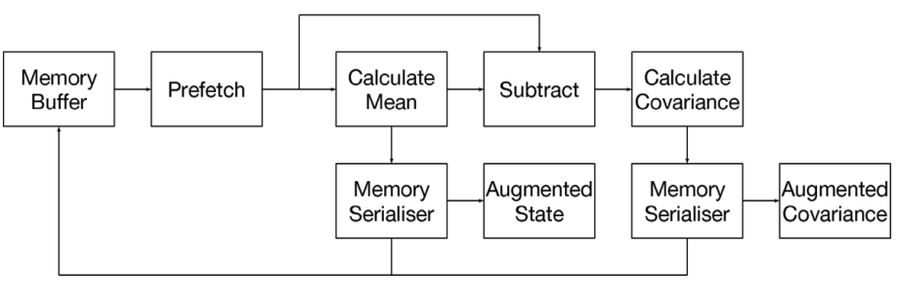


Figure 8.
Block diagram of the predict step.

The processor may initiate a `predict` step once it has placed valid transformed sigma 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 which is also the a priori state estimate (10). The transformed sigma points and the mean are then used to calculate the 'sigma point residuals' via a `subtract` operation. From the 'sigma point residuals', the covariance of the set of transformed sigma points is calculated which is also the a priori covariance (11). Calculation of the mean and covariance is implemented by the calculate mean/covariance module described in Section 3.3.1; this section also describes the details of the 'sigma point residuals'. Once the calculations are complete, the IP core writes the a priori state and covariance to the memory buffer so that both the processor and IP core have the current state estimate. Once the `predict` step is completed, a control bit is set to notify the processor, and, if included, an interrupt generated.

3.3.1 Calculation of mean/covariance

Calculating the mean and covariance of the transformed sigma points is both very similar, meaning both can be calculated by the same datapath. Consider the calculation for the mean of the `predict` step transformed sigma points:

$$\hat{\mathbf{x}}_k^- = \sum_{i=1}^{N} W_i \boldsymbol{\chi}_i^x \tag{25}$$

This is a simple column-wise multiply-accumulation. Consider the calculation of the covariance:

$$\mathbf{P}_k^- = \sum_{i=1}^{N} W_i \left[\boldsymbol{\chi}_i^x - \hat{\mathbf{x}}^-\right]\left[\boldsymbol{\chi}_i^x - \hat{\mathbf{x}}^-\right]^T \tag{26}$$

The subtraction looks like it will cause inefficiencies in the datapath, similar to the division operation in the original Cholesky decomposition. However, let $\tilde{\boldsymbol{\chi}}_i = \boldsymbol{\chi}_i^x - \hat{\mathbf{x}}^-$ be the ith column of $\tilde{\boldsymbol{\chi}}$, and then the covariance calculation reduces to

$$\mathbf{P}_k^- = \sum_{i=1}^{N} W_i \tilde{\boldsymbol{\chi}}_i \tilde{\boldsymbol{\chi}}_i^T \tag{27}$$

This 'sigma point residual' matrix $\tilde{\boldsymbol{\chi}}$ is of size $M_{state} \times N$ where M_{state} is the number of state variables and $N = M + 2$ is the number of sigma points. The element-wise calculation is then

$$P_{ij}^- = \sum_{k=1}^{N} W_1 \tilde{\chi}_{ik} \tilde{\chi}_{jk} \tag{28}$$

This expression involves two multiplications followed by an accumulation; if these 'sigma point residuals' are calculated first with a `subtract` operation, then both the mean and covariance calculations simply involve a series of multiplications and accumulation. Similar to the matrix multiply-add operation, the basic calculations are encapsulated into one processing element, and then additional PEs are added to the datapath in order to calculate additional rows in parallel; the datapath can be seen in **Figure 9**.

The input to the datapath is either the transformed sigma points to calculate the mean or the residuals to calculate the covariance. The FIFO is used to skip the first

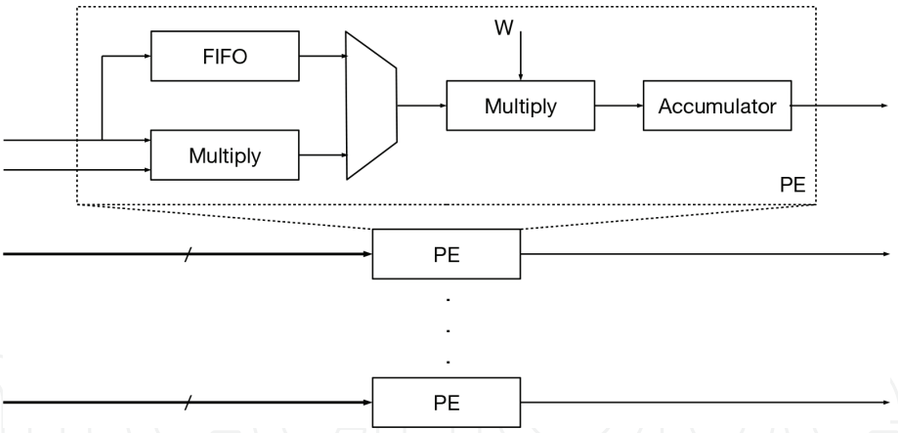


Figure 9.
Calculate mean/covariance operation. W refers the sigma point weights W_0, W_1.

multiplication when calculating the mean; the multiplexer selects which value is calculated.

3.4 Update step

The `update` step corrects the a priori state estimate with a set of observations to generate the new state estimate. Many of the calculations in the `update` step are very similar to the `predict` step; the architecture for the `update` step can be seen in **Figure 10** showing the data flow between modules.

As with the `predict` step, the processor must first place the valid transformed sigma points into the memory buffer before signalling the IP core to begin. First, the prefetch module converts the transformed sigma points into a parallel memory structure. The mean and 'sigma point residuals' are calculated and then used to calculate the observation covariance. The `update` 'sigma point residuals' are also combined with the `predict` 'sigma point residuals', which were calculated during the `predict` step, to calculate the cross-covariance between the two system models. The observation residual, $\tilde{\mathbf{z}} - \hat{\mathbf{z}}$ (17), is calculated with the current set of observations in the memory buffer. The observation and cross-covariance are used to calculate the Kalman gain before the matrix multiply-add modules use the Kalman gain and the a priori state estimate and covariance to calculate the new state estimate and covariance. The new estimates overwrite the a priori 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, setting a control bit and/or generating an interrupt.

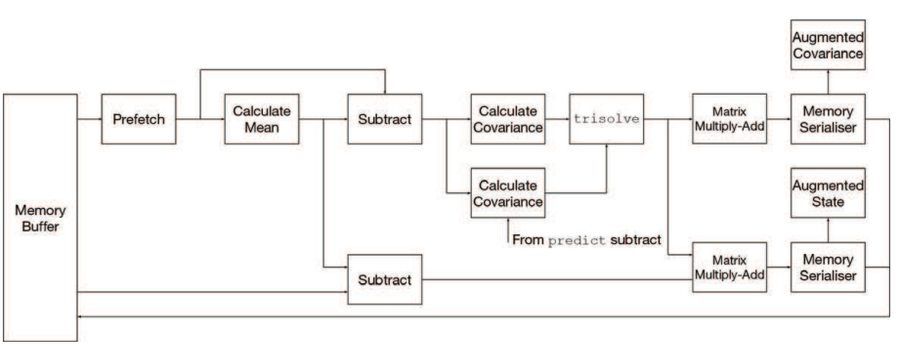


Figure 10.
Block diagram of the update step.

4. Testing and validation of the hardware/software codesign

To validate the implementation of the hardware/software UKF and demonstrate its effectiveness, an example application is presented. This demonstration emulates the attitude determination subsystem of a single, uncontrolled nanosatellite. It is envisioned that a system designer looking to use the HW/SW UKF in a new application simply formulates the UKF appropriately for that application, i.e. formulates the system models (1), (2) and sets the algorithm parameters. Then, once the UKF algorithm has been defined, the HW/SW codesign detailed in Section 3 can then be used to actually implement the UKF and accelerate its performance. The example application attempts to employ this process.

The UKF was implemented using a number of methods for validation and comparison purposes. Once formulated, the UKF was first implemented in MATLAB (SW) to validate the design of the UKF algorithm. Next, the UKF was implemented again using the HW/SW codesign on an FPGA development board in order to validate the codesign. Finally, the UKF was implemented a third time in C (SW), but on the same FPGA development board, to provide a performance benchmark the HW/SW codesign could be compared to.

The FPGA development board used was the Zedboard, featuring a Xilinx Zynq-7000 series XC7Z020, seen in **Figure 11**. The relevant features of the board are:

- Dual ARM Cortex-A9 processor system (PS) @ 667 MHz
- The equivalent of an Artix-7 device in programmable logic (PL)
- AXI4 PS-PL interface

The HW/SW codesign was implemented for the 1 PE and 2 PE cases. The hardware part of the codesign, the IP core, was developed in Verilog and synthesised and implemented using Vivado 2014.1; basic arithmetic was implemented using floating point IP cores from Xilinx's IP catalogue. All designs used a single-precision (IEEE 754–2008) number representation. The target synthesisable frequency for the IP core was 100 MHz, and the 2 PE case instantiated two processing elements for the whole design (i.e. each individual module had two processing elements). The software part of the codesign was implemented in C as bare-metal application on the processor system. The general-purpose AXI4

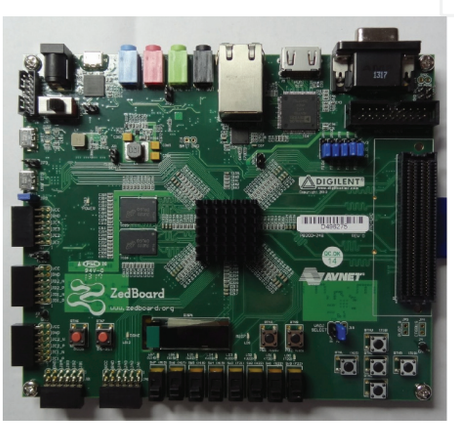


Figure 11.
Zedboard development board used for two of the UKF implementations.

interface between the PS and the PL was used by the two parts to communicate with each other (@ 100 MHz as well). The C (SW) implementation of the UKF was a bare-metal application that used the GNU Scientific Library (GSL) for its vector and matrix manipulations. All software was compiled using the -O2 optimisation flag.

To test the different UKF implementations, a simulator was constructed in MATLAB to model the nanosatellite's motion; the details are given in Section 4.5. Simulated sensor measurements were generated from the nanosatellites' motion and passed to each of the three UKF implementations, which act as the attitude determination subsystem. For the MATLAB implementation, the simulated measurements could be passed directly. For the HW/SW codesign and the C (SW) implementations, the simulated measurements were first exported to a C header which was included during compilation.

4.1 System model

The nanosatellite is modelled as a 1 U CubeSat. The attitude of the nanosatellite is represented by the unit quaternion $q = [\mathbf{q}, q_0]^T$ where $\mathbf{q} = [q_1, q_2, q_3]^T$ and which satisfies $q_1^2 + q_2^2 + q_3^2 + q_0^2 = 1$.

The kinematic equations for the satellite in terms of quaternions are given by

$$\dot{\mathbf{q}} = \frac{1}{2}(q_0 I_{3\times 3} + \mathbf{q}^{\times})\boldsymbol{\omega} \tag{29}$$

$$\dot{q}_0 = -\frac{1}{2}\mathbf{q}^T\boldsymbol{\omega} \tag{30}$$

where $\boldsymbol{\omega}$ is the angular rate and $\mathbf{q}^{\times}$ is the skew-symmetric matrix of $\mathbf{q}$ given by

$$\mathbf{q}^{\times} = \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix} \tag{31}$$

4.2 Sensor model

We consider a basic sensor set common on nanosatellites—a three-axis MEMS IMU including an accelerometer, gyroscope and magnetometer. We use the standard gyroscopic model for the gyroscope:

$$\mathbf{z}_g = \boldsymbol{\omega}_T + \boldsymbol{\beta} + \boldsymbol{\eta}_g \tag{32}$$

$$\dot{\boldsymbol{\beta}} = \boldsymbol{\eta}_d \tag{33}$$

where $\boldsymbol{\omega}_T$ is the true angular velocity, $\boldsymbol{\beta}$ is the gyroscopic bias, $\dot{\boldsymbol{\beta}}$ is the gyroscopic bias drift and $\boldsymbol{\eta}_g$, $\boldsymbol{\eta}_d$ are noise terms that are assumed to be zero-mean Gaussians. Similarly, we model the accelerometer and magnetometer as

$$\mathbf{z}_a = \mathbf{a}_T + \boldsymbol{\eta}_a \tag{34}$$

$$\mathbf{z}_m = \mathbf{m}_T + \boldsymbol{\eta}_m \tag{35}$$

where $\mathbf{a}_T$ is the true local acceleration vector, $\mathbf{m}_T$ is the true local magnetic vector and $\boldsymbol{\eta}_a$, $\boldsymbol{\eta}_m$ are, again, zero-mean Gaussian measurement noise terms.

4.3 Predict model

We use a dead-reckoning model and the gyroscopic data to predict the motion of the nanosatellite. However, it is necessary to account for the gyroscopic bias drift, so we estimate the current gyroscopic bias as well. Let the state vector be

$$\mathbf{x} = \left[\mathbf{q}, q_0, \boldsymbol{\beta}\right]^T \tag{36}$$

The predict model, f, is then

$$f\left(\boldsymbol{\chi}^x_{k-1|k-1}, \boldsymbol{\chi}^w_{k-1|k-1}\right) = \boldsymbol{\chi}^x_{k-1|k-1} + f'\left(\boldsymbol{\chi}^x_{k-1|k-1}, \boldsymbol{\chi}^w_{k-1|k-1}\right) \cdot dt \tag{37}$$

$$f'\left(\boldsymbol{\chi}^x_{k-1|k-1}, \boldsymbol{\chi}^w_{k-1|k-1}\right) = \begin{bmatrix} \frac{1}{2}\left(q_0 I_{3\times 3} + \mathbf{q}^{\times}\right)\mathbf{z}_g \\ -\frac{1}{2}\mathbf{q}^T\mathbf{z}_g \\ \mathbf{0}_{3\times 1} \end{bmatrix} + \mathbf{w}_k \tag{38}$$

where dt is the time step between samples, $\mathbf{w}_k = \left[\boldsymbol{\eta}_q, \dot{\boldsymbol{\beta}}\right]^T$ is the process noise and $\boldsymbol{\eta}_q$ is assumed to be a zero-mean Gaussian.

4.4 Update model

The accelerometer and magnetometer data are used to correct for the gyroscopic bias, so the observation model, h, is

$$h\left(\boldsymbol{\chi}^x_{k-1|k-1}, \boldsymbol{\chi}^v_{k-1|k-1}\right) = \begin{bmatrix} A_q(\mathbf{q})g\mathbf{b}_a \\ A_q(\mathbf{q})\mathbf{b}_m \end{bmatrix} + \mathbf{v}_k \tag{39}$$

where $\mathbf{b}_a$ and $\mathbf{b}_m$ are the respective body frame vectors, g is the magnitude of the gravity vector (assumed 8.94 m.s^{-2} at an altitude of 300 km), $\mathbf{v}_k = [\boldsymbol{\eta}_a, \boldsymbol{\eta}_m]$ is the measurement noise and $A_q(\mathbf{q})$ is the rotation matrix between the body frame and the local frame given by

$$A_q(\mathbf{q}) = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_0q_2 + q_1q_3) \\ 2(q_1q_2 + q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_0q_1 + q_2q_3) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} \tag{40}$$

4.5 Simulation model

Collecting all the relevant terms, the initial augmented state vector is given by

$$\mathbf{x}^a_0 = \left[\mathbf{q}, q_0, \boldsymbol{\beta}, \mathbf{0}_{4\times 1}, \mathbf{0}_{3\times 1}, \mathbf{0}_{3\times 1}, \mathbf{0}_{3\times 1}\right]^T, \tag{41}$$

and the initial augmented covariance is a diagonal matrix with diagonal terms:

$$\mathrm{diag}\left(\mathbf{P}^a_0\right) = \left[\mathbf{1}_{6\times 1}, \boldsymbol{\eta}_q, \dot{\boldsymbol{\beta}}, \boldsymbol{\eta}_a, \boldsymbol{\eta}_m\right] \tag{42}$$

The state vector length is 7, the number of observation variables is 6 and the augmented state vector length is 20. The quaternion noise term was modelled with covariance $\boldsymbol{\eta}_q = 10^{-6}$. The simulated sensor set was homogeneous, so the modelled

errors are the same for each nanosatellite. The gyroscopic bias drift was modelled with covariance $\boldsymbol{\eta}_d = 10^{-2^\circ}/s^2$. The measurement noise terms were modelled with covariances: $\boldsymbol{\eta}_g = 10^{-1^\circ}/s$, $\boldsymbol{\eta}_a = 10^{-2}g$, $\boldsymbol{\eta}_m = 10^{-2}\,gauss$. The satellite was modelled as undergoing slow tumbling. The motion was modelled using the Euler angles in a local ground frame, which is relevant in most remote sensing applications; here, we use roll-pitch-yaw to refer to rotations about the x-y-z axis, respectively.

To generate the sensor measurements, the simulated motions were converted into the body frame via rotation matrix with 1–2-3 referring to roll-pitch-yaw, respectively:

$$A_{euler} = \begin{bmatrix} c_1c_2 & c_1s_2s_3 - s_1c_3 & s_1s_3 + c_1s_2c_3 \\ s_1c_2 & s_1s_2s_3 + c_1c_3 & s_1s_2s_3 - c_1s_3 \\ -s_2 & c_2s_3 & c_2c_3 \end{bmatrix} \tag{43}$$

It is assumed that the magnetometer is aligned with the x-axis ($\mathbf{b}_m = [1, 0, 0]$) and the accelerometer is aligned with the z-axis ($\mathbf{b}_a = [0, 0, 1]$). Next, using the sensor models described earlier, noise terms were added to the sensor 'truth' data which was then sampled at 1 Hz to simulate measurements from an actual set of sensors.

4.6 Results

The UKF was simulated in MATLAB environment as well as on the Zedboard development board. For the Zedboard implementations, the simulated sensor data set was loaded into the onboard memory (RAM) and the UKF simulated as if it were receiving data from the actual sensors. The data set used in all three implementations was the same. State estimates from the UKF were stored on the Zedboard for the duration of the simulation and then read back into MATLAB afterwards for analysis.

All three implementations produced (within working precision) the simulation results in **Figure 12a** and **b**; these figures show the absolute attitude error (i.e. the difference between the UKF estimated attitude and the simulated 'truth') of the nanosatellite. In **Figure 12a**, the top graph shows the first tenth of a second of the simulation, highlighting early convergence of the filter to the truth from an

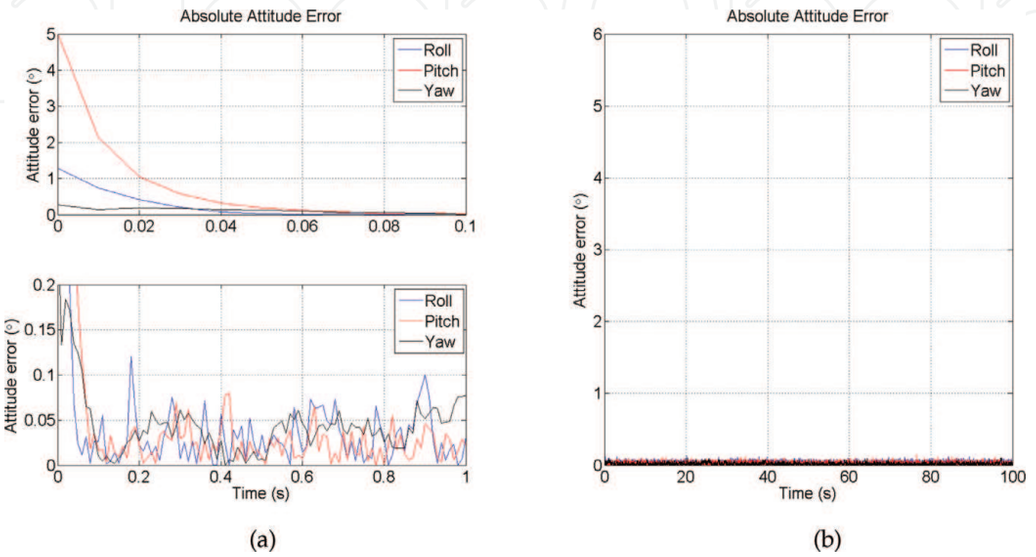


Figure 12.
Absolute attitude error.

	SW	1 PE	2 PE
Total	660	363	272

Table 1.
Overall latency for the single nanosatellite. All values in μs.

initial noisy estimate. The bottom figure shows the first second of the simulation, highlighting the ability of the filter to maintain its accuracy ($< 0.1^{\circ}$ error) after convergence. **Figure 12b** shows that the UKF is able to correct for the inaccuracies arising from the gyroscopic bias and bias drift over the full duration of the simulation.

(a) At the very beginning of the simulation.

(b) For the full simulation.

These results demonstrate that there are no implementation issues when taking the UKF to a HW/SW codesign; the codesign, and IP core, is able to completely replicate software-based implementations of the UKF. The overall latency of the C (SW) implementation and the HW/SW codesign (serial and parallel) for the single nanosatellite case were measured using the ARMv7 Performance Monitoring Unit (PMU) and can be seen in **Table 1**. This overall latency is the time taken to complete one full iteration of the UKF (all steps). The 1 PE case offers a modest 1.8× increase in performance over the C (SW) implementation and can be run at ≈2 kHz which is more than adequate for the sampling frequency assumed by the simulation. The 2 PE case offers a slightly better 2.4× speed-up over the C (SW) implementation. Note that the processor system operates at a clock frequency more than six times the frequency used by the IP core (667 vs. 100 MHz), yet the IP core is still able to outperform the C (SW) implementation.

5. Implementation analysis of the hardware/software codesign

Synthesis and implementation runs were targeted at the Zynq-7000 XC7Z045 at a target frequency of 100 MHz. Though the implementations of the example applications presented in Section 4 was for the Zynq-7000 XC7Z020, the codesign does not fit on this device for larger numbers of processing elements. In order to still compare implementation details, this larger device in the Zynq-7000 family is used instead. All the devices in the Zynq-7000 family feature the same processing system; the only difference for larger devices is the amount of programmable logic available.

Resource utilisation of the device by the IP core is reported by Vivado post-implementation. The power analysis is done via the Xilinx Power Estimator (XPE) post-implementation; all power estimates exclude the device static power dissipation and the processing system power draw.

The execution time (latency) for the hardware part is measured via behavioural simulation in Vivado Simulator, assuming a clock frequency of 100 MHz; this assumption was validated post-implementation for all designs. Though behavioural simulations are usually used for only functional verification, Vivado Simulator provides cycle-accurate execution times as long as timing assumptions made in the simulation are verified post-implementation. The entire IP core utilises synchronous logic and is on a single clock domain which makes confirming the proper distribution of the assumed clock signals, in this case 100 MHz, relatively straightforward.

The execution time (latency) of the software part is measured via the ARMv7 Performance Monitoring Unit (PMU), which counts processor clock cycles between

two epochs; because the number of processor clock cycles to perform a given task can vary, each measurement was conducted at least 10 times, and the average latency measured is reported here.

5.1 Synthesis results

Synthesis results for a selection of different numbers of processing elements can be seen in **Table 2**. These results do not include the processor but do include the logic necessary for the AXI4 interface ports. The initial numbers of PEs were chosen to be multiples of the number of augmented state variables so that the major datapaths remained data efficient. Recall, for example, the matrix multiply-add datapath (Section 3.2.2); each PE calculates an entire row in the result matrix. If the number of PEs is not a multiple of the size of the matrix, then the last iteration of the calculations will not have enough data to fill all the PEs making the datapath slightly inefficient.

For low numbers of processing elements, the codesign utilises a relatively small percentage of the available resources. The XC7Z045 is a mid-range device in the Zynq-7000 series which means even the 10 PE case still only uses a quarter of the available LUTs. The codesign does not require a proportionally large amount of any one resource; in fact, the design uses a disproportionately smaller amount of FFs than other resources. This will allow easier integration into a full SoC, particularly if partially reconfigurable regions are used. Requiring too much of any one resource type can lead to placement and routing issues since resource on-chip locations are fixed by the manufacturer. This also implies that additional register stages could be added to major datapaths, which would increase the overall latency but could allow an increase in clock frequency as well. If the increase in clock frequency was greater than the increase in latency, the overall performance of the design would benefit.

5.2 Power consumption

A power consumption breakdown for the hardware IP core (i.e. excluding the processor) can be seen in **Table 3**. The power consumption for low numbers of PEs is reasonably low, due to the area efficiency design goals and the heavy utilisation of the FPGA clock that enable resources to disable modules that are not currently in use. For reference, the device static power consumption (@ 25°C) is ≈245 mW, and the rough power consumption of the processing system is ≈1.5 W. A conservative estimate of the electrical power available to a CubeSat is in the order of 1–2 W per unit [15]; larger 2–3 U or more CubeSats have a greater surface area to cover in solar panels. The 1 PE case could be incorporated into a 1 U or larger CubeSat with relative ease, but even for just the 2 PE case, a 2 U CubeSat or larger may be necessary.

Resource	1 PE	2 PE	5 PE	10 PE
FF	7668 (2)	14,286 (3)	27,311 (6)	48,714 (11)
LUT	5764 (3)	15,158 (7)	29,500 (13)	53,427 (24)
BRAM	16.5 (3)	36.5 (7)	62 (11)	109.5 (20)
DSP48	35 (4)	62 (7)	104 (12)	182 (20)

Table 2.
Resource utilisation (% total) on the XC7Z045.

Resource	1 PE	2 PE	5 PE	10 PE
Clocks	38	74	136	234
Signals	24	83	144	261
Logic	23	76	126	219
BRAM	51	82	112	209
DSP	4	6	21	52
Total	140	336	549	975

Table 3.
Power consumption of the codesign. All values in mW.

	SW	1 PE	2 PE	5 PE	10 PE
Sig. gen.	—	—	92	61	51
System model	52	137	137	137	137
Predict	522	170	13	8.5	6.5
Update	87	56	30	21	17
Total	660	363	272	228	212

Table 4.
Latency of each stage for the codesign. System models encompass propagation through both the `predict` *and the* `update` *models on the processor. All values in μs.*

5.3 Timing analysis

A breakdown of the execution time (latency) of different modules can be seen in **Table 4**. The design spends a large amount of the time propagating the sigma points through the two system models. The majority of the time spent by the design is actually in these system models, making the software part the main bottleneck. Looking at the sigma point propagation process a little closer, however, the latency of reading the sigma points from the memory buffer and of writing the transformed points back to the memory buffer was 116 μs. The actual calculation of the system models took a mere 21 μs. So, the bottleneck is actually the speed of the AXI4 port in transferring data between the processor and the memory buffer. Using a higher-performance communication, bus or other techniques such as direct memory access (DMA) ports may alleviate this issue, but intra-chip communication methods are beyond the scope of this chapter.

For the hardware part, the majority of time is spent in the `sig_gen` step. The two modules in the `sig_gen` step, the triangular linear equation solver and the matrix multiply-add, are both large matrix operations which scale with the number of augmented state variables. Operations in the `predict` and `update` steps tend to scale with the number of state or observation variables, respectively, which are always necessarily smaller than the number of augmented state variables. It should be noted that the hardware part appears to suffer from diminishing returns with regard to decreasing the latency as the number of processing elements increases.

6. Conclusion

In this chapter, a scalable FPGA-based implementation of the unscented Kalman filter was presented. The proposed design balances development effort/complexity

with performance, combining the advantages of both the traditional software approach and hardware approaches to create a design that system designers can easily use in a potentially wide variety of applications. Simulation and physical implementation results of the codesign were presented. The demonstration application simulated the attitude determination system of an uncontrolled nanosatellite, and the physical implementation was performed on the Xilinx XC7Z045.

IntechOpen

IntechOpen

Author details

Jeremy Soh and Xiaofeng Wu*
School of Aerospace, Mechanical and Mechatronic Engineering, University of Sydney, Sydney, Australia

*Address all correspondence to: xiaofeng.wu@sydney.edu.au

IntechOpen

© 2018 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. CC BY

References

[1] Soh J. A scalable, portable, FPGA-based implementation of the Unscented Kalman Filter. Ph.D. University of Sydney, Feb. 2017. Available from: http://hdl.handle.net/2123/17286

[2] Michell GD, Gupta RK. Hardware/software co-design. In: Proceedings of the IEEE 85.3; Mar. 1997. pp. 349-365. ISSN: 0018-9219. DOI:10.1109/5.558708

[3] Wolf W. A decade of hardware/software codesign. In: Computer 36.4; Apr. 2003. pp. 38-43. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1193227

[4] Teich J. Hardware/software Codesign: The past, the present, and predicting the future. In: Proceedings of the IEEE 100.Special Centennial Issue; May 2012. pp. 1411-1430. ISSN: 0018-9219. DOI: 10.1109/JPROC.2011.2182009

[5] Gelb A. Applied Optimal Estimation. Cambridge, Massachusetts: MIT Press; 1974

[6] Nørgaard M, Poulsen NK, Ravn O. "New developments in state estimation for nonlinear systems". Automatica 36.11 (2000), pp. 1627-1638. ISSN: 0005-1098. DOI: http://dx.doi.org/10.1016/S0005-1098(00)00089-3. URL: http://www.sciencedirect.com/science/article/pii/S0005109800000893

[7] Simon D. Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches. Hoboken, New Jersey: John Wiley & Sons; 2006

[8] Crassidis JL, Markley FL, Cheng Y. Survey of nonlinear attitude estimation methods. Journal of Guidance Control and Dynamics. 2007;**30**(1):12

[9] Patwardhan SC, Narasimhan S, Jagadeesan P, Gopaluni B, Shah SL. Nonlinear Bayesian state estimation: A review of recent developments. In: Control Engineering Practice 20.10; 2012. 4th Symposium on Advanced Control of Industrial Processes (ADCONIP). pp. 933-953. ISSN: 0967-0661. DOI: http://dx.doi.org/10.1016/j.conengprac.2012.04.003. URL: http://www.sciencedirect.com/science/article/pii/S0967066112000871

[10] Julier S, Uhlmann J. Unscented filtering and nonlinear estimation. Proceedings of the IEEE. 2004;**92**(3): 401-422

[11] Kandepu R, Foss B, Imsland L. "Applying the unscented Kalman filter for nonlinear state estimation". Journal of Process Control 18.7–8 (2008), pp. 753-768. ISSN: 0959-1524. DOI: 10.1016/j.jprocont.2007.11.004. URL: http://www.sciencedirect.com/science/article/pii/S0959152407001655

[12] Julier S. The spherical simplex unscented transformation. American Control Conference, 2003. Proceedings of the 2003. Vol. 3. June 2003. pp. 2430-2434. DOI: 10.1109/ACC.2003.1243439

[13] Golub GH, Van Loan CF. Matrix computations. 3rd ed. Baltimore: Johns Hopkins University Press; 1996

[14] Yang D, Peterson G, Li H, Sun J. An FPGA implementation for solving Least Square problem. In: Field Programmable Custom Computing Machines, 2009. FCCM '09. 17th IEEE Symposium on. Apr. 2009, pp. 303-306. DOI: 10.1109/FCCM.2009.47

[15] Selva D, Krejci D. A survey and assessment of the capabilities of Cubesats for earth observation. Acta Astronautica. 2012;**74**:50-68. ISSN: 0094-5765. DOI: http://dx.doi.org/10.1016/j.actaastro.2011.12.014. URL: http://www.sciencedirect.com/science/article/pii/S0094576511003742