A Dual State Ensemble Kalman Filter for Hydrologic Models

by

William John Cook

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Science in Computer Science

at the

UNIVERSITY OF MONTANA

June 2019

Author .....
Department of Computer Science
May 30, 2019

Certified by .....
William J. Dally
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Dual State Ensemble Kalman Filter for Hydrologic Models

by

William John Cook

Submitted to the Department of Computer Science
on May 30, 2019, in partial fulfillment of the
requirements for the degree of
Masters of Science in Computer Science

Abstract

In this thesis, I will discuss Dual State Ensemble Kalman Filters and how they can be applied to Hydrologic models.

Thesis Supervisor: William J. Dally
Title: Associate Professor

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Symbols

$\mathrm{dog} = \mathrm{dog}$

Chapter 1

Introduction

1.1 Background

This research is part of the **daWUAP** project led by Marko Maneta. The **daWUAP** team has created a hydrological rainfall-runoff model that predicts streamflows across the state of Montana. **daWUAPhydroengine** is informed by a variety of smaller models, including a groundwater model, snow water equivalient (swe) model, and agricultural component model. Despite this complexity, **daWUAPhydroengine** is designed to be a quick and efficient model.

This paper focuses on the calibration of **daWUAPhydroengine** via a Dual State Parameter Estimation Ensemble Kalman Filter. Uniquely, this filter operates on multiple high dimensional parameters represented as raster images. These raster images are both ingested and outputted by the **daWUAPhydroengine** and were previously set at a constant value. The filter aims to both calibrate and distribute these variables geospatially.

1.2 Motivation

Hydrological models generate various states that are generally determined by a set of time-invariant parameters. In practice these parameters are calibrated continuously and are tweaked as soon as new observations become available. Parameter calibration

techniques have recently been the subject of research [8] [7], although many of the earliest calibration methods are unable to account for all sources of error [1]. More recently, sequential parameter estimation through techniques such as the Ensemble Kalman Filter have been developed that can 1) simultaneously calibrate parameter and state estimates and 2) take all sources of uncertainty into account [2].

The Kalman Filter is an efficient model for continuous parameter estimation because it is a sequential data assimilation algorithm, only needing the previous timestep's state estimate, parameter estimate, and co-variance matrices to update the current timestep's state estimate, parameter estimate, and co-variance matrices. The more advanced Ensemble Kalman Filter can account for non-linearity and replaces the priori covarience matrix used in other kalman filtering methods such as the extended or unscented kalman filter with an ensemble covariance matrix, allowing for more efficient computation of large state vectors.

Research has been done on applying dual state parameter estimation ensemble Kalman filtering to hydrologic models [6]. Further research into the calibration of hydrologic parameters, particularly geospatial parameters, will help both inform future hydrologic models and allow for examination of the effectiveness of different techniques that distribute geospatial parameters across a landscape.

1.3 Design Constraints

The optimization of the parameters must be done in a way that is reasonably efficient. Since models such as **daWUAPhydroengine** ingest and output dense raster data, for example, many simpler calibration implementations will become unwieldy because of large co-variance matrices. Furthermore, since **daWUAPhydroengine** is designed to be a quick and efficient model, it is preferable that any calibration component is similarly efficient.

1.4 Contributions

The main contribution of this paper is a new framework and implementation for calibration of large raster parameters via an ensemble kalman filter. It also covers the real life case application of this framework to the **daWUAP** hydrologic engine to validate these methods. This allows major obstacles to be observed. The code is modular and, with some modification, be usable with any Bayesian model.

1.5 Thesis Organization

Section 2 covers the theory behind a Dual State Ensemble Kalman Filter's implementation. Section 3 discusses the unique changes made to the DSKE parameter sampling methods. Section 4 covers the application of our DSEKF to **daWUAPhydroengine**.

Chapter 2

Theory

2.1 The History of Kalman Filters

R.E Kalman published the article *A New Approach to Linear Filtering and Prediction Problems* in 1960 [5]. Since then, the so-called "Kalman Filter" has been tested, researched, and improved extensively. Kalman's original algorithm was limited to linear systems. The development of the Extended Kalman Filter allowed Kalman Filters to operate on non-linear systems with some limitations. More recently, the Unscented Kalman Filter [4] and the Ensemble Kalman Filter [1] have been developed to work on non-linear systems.

2.2 The Linear Kalman Filter

The original Kalman filter was created to solve problems where both a predictive sequential model and a series of observations is available. The predictive model can be represented as the linear stochastic difference equation

$$x_i = Ax_{i-1} + Bu_{i-1} + w_{i-1} \tag{2.1}$$

Where A is the model matrix which serves to transform the vector x_{i-1} to the current timestep, B is the control matrix that transforms the control vector u_i to account for external forces on the model, w_i is a vector of model error, and i is the timestep.

An observation for any given timestep i can be represented as

$$z_i = Hx_i + v_i \tag{2.2}$$

where z_i is the vector of observations, x_i is the vector of true states, H is a masking matrix, and v_i is a vector of measurement errors. w_i and v_i are assumed to be independent, normally distributed random variables with probability distributions defined by

$$P(w) \sim N(0, Q) \tag{2.3}$$

$$P(v) \sim N(0, R) \tag{2.4}$$

2.2.1 Algorithm

Kalman filters optimize model predictions by blending predicted states with that timestep's observations. Conveniently, the algorithm's steps are separated into *prediction* and *update* categories. The initial prediction algorithm (2.5) obtains the current timestep's vector of states using the same equation as (2.1) with the removal of the random unknown vector w. To track the effects of ignoring w the prior error covariance matrix P^- is calculated (2.6).

Equation (2.8) returns the updated prediction $\hat{x}_i^+$ by multiplying the innovation between the observation and the masked prediction by the kalman gain K, which is defined in (2.7). Finally, the error covariance matrix is updated in (2.9) to reflect the

Table 2.1: Prediction Equations - Discrete Kalman Filter

Name	Equation
Model Prediction	$\hat{x}_i^- = A\hat{x}_{i-1}^+ + Bu_{i-1}$ (2.5)
Update Prior Covariance	$P_i^- = AP_i^+A^T + Q$ (2.6)

more accurate nature of the updated prediction.

Table 2.2: Update Equations - Discrete Kalman Filter

Name	Equation
Kalman Gain	$K_i = P_i^- H^T (HP_i^- H^T + R)^{-1}$ (2.7)
Update Estimate	$\hat{x}_i^+ = \hat{x}_i^- + K_i(z_i - H\hat{x}_i)$ (2.8)
Update Posterior Covariance	$P_i^+ = (I - K_i H)P_i^-$ (2.9)

2.3 The Whatever We call It Ensemble Kalman Filter

The Dual Ensemble Kalman Filter is a sequential data assimilation method. It can be split into four different subsections: The initial prediction phase, the parameter correction phase, the second prediction phase, and the state correction phase.

2.3.1 The Prediction Phase

According to Jazwinski [3] any discrete nonlinear stochastic model can be defined as:

$$x_{t+1} = f(x_t, u_t, \theta_t) + \omega_t \tag{2.10}$$

where x_t is an n dimensional vector representing the state variables of the model at time step t, u_t is a vector of forcing data (e.g temperature or precipitation) at time step t, and θ_t is a vector of model parameters which may or may not change per time step (e.g soil beta or DDF). The non-linear function f takes these variables as inputs. The error variable ω_t accounts for both model structural error and for any uncertainty in the forcing data.

Chapter 3

Example

Micro-optimization is a technique to reduce the overall operation count of floating point operations. In a standard floating point unit, floating point operations are fairly high level, such as "multiply" and "add"; in a micro floating point unit (μFPU), these have been broken down into their constituent low-level floating point operations on the mantissas and exponents of the floating point numbers.

Chapter two describes the architecture of the μFPU unit, and the motivations for the design decisions made.

Chapter three describes the design of the compiler, as well as how the optimizations discussed in section 3.2 were implemented.

Chapter four describes the purpose of test code that was compiled, and which statistics were gathered by running it through the simulator. The purpose is to measure what effect the micro-optimizations had, compared to unoptimized code. Possible future expansions to the project are also discussed.

3.1 Motivations for micro-optimization

The idea of micro-optimization is motivated by the recent trends in computer architecture towards low-level parallelism and small, pipelineable instruction sets [**?**, **?**]. By getting rid of more complex instructions and concentrating on optimizing frequently used instructions, substantial increases in performance were realized.

Another important motivation was the trend towards placing more of the burden of performance on the compiler. Many of the new architectures depend on an intelligent, optimizing compiler in order to realize anywhere near their peak performance [**?**, **?**, **?**]. In these cases, the compiler not only is responsible for faithfully generating native code to match the source language, but also must be aware of instruction latencies, delayed branches, pipeline stages, and a multitude of other factors in order to generate fast code [**?**].

Taking these ideas one step further, it seems that the floating point operations that are normally single, large instructions can be further broken down into smaller, simpler, faster instructions, with more control in the compiler and less in the hardware. This is the idea behind a micro-optimizing FPU; break the floating point instructions down into their basic components and use a small, fast implementation, with a large part of the burden of hardware allocation and optimization shifted towards compile-time.

Along with the hardware speedups possible by using a μFPU, there are also optimizations that the compiler can perform on the code that is generated. In a normal sequence of floating point operations, there are many hidden redundancies that can be eliminated by allowing the compiler to control the floating point operations down to their lowest level. These optimizations are described in detail in section 3.2.

3.2 Description of micro-optimization

In order to perform a sequence of floating point operations, a normal FPU performs many redundant internal shifts and normalizations in the process of performing a sequence of operations. However, if a compiler can decompose the floating point operations it needs down to the lowest level, it then can optimize away many of these redundant operations.

If there is some additional hardware support specifically for micro-optimization, there are additional optimizations that can be performed. This hardware support entails extra "guard bits" on the standard floating point formats, to allow several

unnormalized operations to be performed in a row without the loss information[1]. A discussion of the mathematics behind unnormalized arithmetic is in appendix **??**.

The optimizations that the compiler can perform fall into several categories:

3.2.1 Post Multiply Normalization

When more than two multiplications are performed in a row, the intermediate normalization of the results between multiplications can be eliminated. This is because with each multiplication, the mantissa can become denormalized by at most one bit. If there are guard bits on the mantissas to prevent bits from "falling off" the end during multiplications, the normalization can be postponed until after a sequence of several multiplies[2].

As you can see, the intermediate results can be multiplied together, with no need for intermediate normalizations due to the guard bit. It is only at the end of the operation that the normalization must be performed, in order to get it into a format suitable for storing in memory[3].

3.2.2 Block Exponent

In a unoptimized sequence of additions, the sequence of operations is as follows for each pair of numbers (m_1,e_1) and (m_2,e_2).

1. Compare e_1 and e_2.
2. Shift the mantissa associated with the smaller exponent $|e_1 - e_2|$ places to the right.
3. Add m_1 and m_2.
4. Find the first one in the resulting mantissa.

[1] A description of the floating point format used is shown in figures **??** and **??**.

[2] Using unnormalized numbers for math is not a new idea; a good example of it is the Control Data CDC 6600, designed by Seymour Cray. [**?**] The CDC 6600 had all of its instructions performing unnormalized arithmetic, with a separate `NORMALIZE` instruction.

[3] Note that for purposed of clarity, the pipeline delays were considered to be 0, and the branches were not delayed.

5. Shift the resulting mantissa so that normalized

6. Adjust the exponent accordingly.

Out of 6 steps, only one is the actual addition, and the rest are involved in aligning the mantissas prior to the add, and then normalizing the result afterward. In the block exponent optimization, the largest mantissa is found to start with, and all the mantissa's shifted before any additions take place. Once the mantissas have been shifted, the additions can take place one after another[4]. An example of the Block Exponent optimization on the expression X = A + B + C is given in figure **??**.

3.3 Integer optimizations

As well as the floating point optimizations described above, there are also integer optimizations that can be used in the μFPU. In concert with the floating point optimizations, these can provide a significant speedup.

3.3.1 Conversion to fixed point

Integer operations are much faster than floating point operations; if it is possible to replace floating point operations with fixed point operations, this would provide a significant increase in speed.

This conversion can either take place automatically or or based on a specific request from the programmer. To do this automatically, the compiler must either be very smart, or play fast and loose with the accuracy and precision of the programmer's variables. To be "smart", the computer must track the ranges of all the floating point variables through the program, and then see if there are any potential candidates for conversion to floating point. This technique is discussed further in section **??**, where it was implemented.

The other way to do this is to rely on specific hints from the programmer that a certain value will only assume a specific range, and that only a specific precision is

[4]This requires that for n consecutive additions, there are $\log_2 n$ high guard bits to prevent overflow. In the μFPU, there are 3 guard bits, making up to 8 consecutive additions possible.

desired. This is somewhat more taxing on the programmer, in that he has to know the ranges that his values will take at declaration time (something normally abstracted away), but it does provide the opportunity for fine-tuning already working code.

Potential applications of this would be simulation programs, where the variable represents some physical quantity; the constraints of the physical system may provide bounds on the range the variable can take.

3.3.2 Small Constant Multiplications

One other class of optimizations that can be done is to replace multiplications by small integer constants into some combination of additions and shifts. Addition and shifting can be significantly faster than multiplication. This is done by using some combination of

$$\begin{aligned} a_i &= a_j + a_k \\ a_i &= 2a_j + a_k \\ a_i &= 4a_j + a_k \\ a_i &= 8a_j + a_k \\ a_i &= a_j - a_k \\ a_i &= a_j \ll m\text{shift} \end{aligned}$$

instead of the multiplication. For example, to multiply s by 10 and store the result in r, you could use:

$$\begin{aligned} r &= 4s + s \\ r &= r + r \end{aligned}$$

Or by 59:

$$\begin{aligned} t &= 2s + s \\ r &= 2t + s \\ r &= 8r + t \end{aligned}$$

Similar combinations can be found for almost all of the smaller integers[5]. [?]

3.4 Other optimizations

3.4.1 Low-level parallelism

The current trend is towards duplicating hardware at the lowest level to provide parallelism[6]

Conceptually, it is easy to take advantage to low-level parallelism in the instruction stream by simply adding more functional units to the μFPU, widening the instruction word to control them, and then scheduling as many operations to take place at one time as possible.

However, simply adding more functional units can only be done so many times; there is only a limited amount of parallelism directly available in the instruction stream, and without it, much of the extra resources will go to waste. One process used to make more instructions potentially schedulable at any given time is "trace scheduling". This technique originated in the Bulldog compiler for the original VLIW machine, the ELI-512. [?, ?] In trace scheduling, code can be scheduled through many basic blocks at one time, following a single potential "trace" of program execution. In this way, instructions that *might* be executed depending on a conditional branch

[5]This optimization is only an "optimization", of course, when the amount of time spent on the shifts and adds is less than the time that would be spent doing the multiplication. Since the time costs of these operations are known to the compiler in order for it to do scheduling, it is easy for the compiler to determine when this optimization is worth using.

[6]This can been seen in the i860; floating point additions and multiplications can proceed at the same time, and the RISC core be moving data in and out of the floating point registers and providing flow control at the same time the floating point units are active. [?]

further down in the instruction stream are scheduled, allowing an increase in the potential parallelism. To account for the cases where the expected branch wasn't taken, correction code is inserted after the branches to undo the effects of any prematurely executed instructions.

3.4.2 Pipeline optimizations

In addition to having operations going on in parallel across functional units, it is also typical to have several operations in various stages of completion in each unit. This pipelining allows the throughput of the functional units to be increased, with no increase in latency.

There are several ways pipelined operations can be optimized. On the hardware side, support can be added to allow data to be recirculated back into the beginning of the pipeline from the end, saving a trip through the registers. On the software side, the compiler can utilize several tricks to try to fill up as many of the pipeline delay slots as possible, as seendescribed by Gibbons. [**?**]

Appendix A

Tables

Table A.1: Armadillos

Armadillos	are
our	friends

Appendix B

Figures

Figure B-1: Armadillo slaying lawyer.

Figure B-2: Armadillo eradicating national debt.

Bibliography

[1] Geir Evensen. Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics. *Journal of Geophysical Research*, 99(C5):10143, may 1994.

[2] Geir Evensen. The Ensemble Kalman Filter: Theoretical formulation and practical implementation. *Ocean Dynamics*, 53(4):343–367, nov 2003.

[3] Andrew H. Jazwinski. Stochastic Processes and Filtering. *Mathematics in Science and Enginerring*, 1970.

[4] Simon J. Julier and Jeffrey K. Uhlmann. New extension of the Kalman filter to nonlinear systems. volume 3068, page 182. International Society for Optics and Photonics, jul 1997.

[5] R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 82(1):35, mar 1960.

[6] Hamid Moradkhani, Soroosh Sorooshian, Hoshin V. Gupta, and Paul R. Houser. Dual state-parameter estimation of hydrological models using ensemble Kalman filter. *Advances in Water Resources*, 28(2):135–147, feb 2005.

[7] Soroosh Sorooshian, Qingyun Duan, and Vijai Kumar Gupta. Calibration of rainfallâĂŘrunoff models: Application of global optimization to the Sacramento Soil Moisture Accounting Model. *Water Resources Research*, 29(4):1185–1194, apr 1993.

[8] Xianhong Xie and Dongxiao Zhang. Data assimilation for distributed hydrological catchment modeling via ensemble Kalman filter. *Advances in Water Resources*, 33(6):678–690, jun 2010.