3-2021

Optimizing a Bank of Kalman Filters for Navigation Integrity

Luis E. Sepulveda

Optimizing Banks of Kalman Filters for Navigation Integrity using Parallel Computing and Efficient Software Design

THESIS

Luis E. Sepulveda, Captain, USAF

AFIT-ENG-MS-21-M-079

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-21-M-079

OPTIMIZING BANKS OF KALMAN FILTERS FOR NAVIGATION INTEGRITY USING PARALLEL COMPUTING AND EFFICIENT SOFTWARE DESIGN

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Luis E. Sepulveda, B.S.E.C.E.

Captain, USAF

March 26, 2021

AFIT-ENG-MS-21-M-079

OPTIMIZING BANKS OF KALMAN FILTERS FOR NAVIGATION INTEGRITY USING PARALLEL COMPUTING AND EFFICIENT SOFTWARE DESIGN

THESIS

Luis E. Sepulveda, B.S.E.C.E.
Captain, USAF

Committee Membership:

Robert C. Leishman, Ph.D
Chair

Scott L. Nykl, Ph.D
Member

Kyle J. Kauffman, Ph.D
Member

AFIT-ENG-MS-21-M-079

Abstract

Alternative navigation is an area of research that employs a variety of sensor technologies to provide a navigation solution in Global Navigation Satellite System degraded or denied environments. The Autonomy and Navigation Technology Center at the Air Force Institute of Technology has recently developed the Autonomous and Resilient Management of All-source Sensors (ARMAS) navigation framework, which utilizes an array of Kalman filters to provide a navigation solution resilient to sensor failures. The Kalman filter array size increases exponentially as system sensors and detectable faults are scaled up, which in turn increases the computational power required to run ARMAS in a real-world application. In an effort to engineer a real-time ARMAS system, this study developed C++ CPU and CUDA GPU versions to examine the performance trade-offs as system sensors and detectable faults are scaled up. Small sensor size configurations proved to be faster on a CPU, while a 3.5 order of magnitude speedup was achieved by the GPU ARMAS implementation over the CPU version on a large sensor system configuration. Also, a novel ARMAS software model is presented that uses layered ARMAS subsystems requiring less processing power than a single ARMAS system architecture instantiation. Results show promise that a real-time ARMAS system can be achieved for large scale applications through parallel processing on a many-core processor architecture and efficient software design.

Acknowledgements

I would like to my advisor, Dr. Leishman, for guiding me through this process and encouraging intellectual fortitude through encountered challenges. My committee for taking the time to refine the concepts and analysis presented in this thesis. Last, but not least, I dedicate this work to my daughter, Elyssa Sepulveda, my son, Evan Sepulveda, and my beautiful wife, Elyana Sepulveda for their support and patience through this entire experience.

Luis E. Sepulveda

Table of Contents

Page

List of Figures

Figure Page

OPTIMIZING BANKS OF KALMAN FILTERS FOR NAVIGATION INTEGRITY USING PARALLEL COMPUTING AND EFFICIENT SOFTWARE DESIGN

I. Introduction

1.1 Background

For the last three decades, the United States Air Force has maintained air dominance in military engagements throughout the world. The ability to strike targets accurately and precisely is dependent on the navigation and timing solutions available to the warfighter. Modern navigation systems rely heavily on Global Navigation Satellite Systems (GNSS) to provide an accurate navigation solution under the assumption that GNSS are available in the area of operations. This is an assumption that may not hold in future contested environments where GNSS transmissions can be degraded or denied by near-peer adversaries. In light of this, the Air Force Institute of Technology's (AFIT) Autonomy and Navigation Technology (ANT) Center is a forward-looking research center seeking to address this type of challenging navigation problem through alternative all-source navigation technology.

Alternative all-source navigation technology is an area of research which employs a variety of sensor technologies to provide a navigation solution in GNSS degraded or denied environments. Recently, the ANT Center has developed the Autonomous and Resilient Management of All-source Sensors (ARMAS) navigation framework, which provides a navigation solution resilient to sensor failures [1]. Configuring GNSS satellites as individual sensors in an ARMAS system can equip warfighters with a coherent navigation solution as they transition and operate into highly contested

environments. Currently, the ARMAS framework is implemented in the MATLAB computing environment for academic research and development.

1.2 Problem Statement

In an effort to transition the academic software into an operational system, the current ARMAS framework must be optimized to meet the real-time constraints of modern navigation systems. In particular, ARMAS requires a nontrival amount of processing power to maintain an array of unique Kalman filters, which identify faulty sensors through a general consensus. The size of the Kalman filter array depends on the number of system sensors and detectable system sensor faults specified by the user. As a general approximation, the Kalman filter array size increases exponentially as system sensors and detectable faults are scaled up linearly, which in turn increases the computational load required to run ARMAS in a real-world application. This thesis answers the over-arching investigative question: How can the academic ARMAS software be optimized towards an operational real-time system?

1.3 Research Objectives

The primary purpose of this research is to explore software design techniques to optimize the execution of at least 10,000 Kalman filter instances within the ARMAS framework. The computational complexity of running the Kalman filter instances will be analyzed to efficiently allocate the computational load across available devices on a heterogeneous platform. The focus will be to exploit collaboration advantages between a Central Processing Unit (CPU) and Graphics Processing Unit (GPU) for efficient workload processing. The goal of this investigation is to engineer software models for different conditions that can scale up while maintaining performance and identify key system criteria that distinguishes the most efficient software model to

use.

1.4 Hypothesis

Acknowledging that developing a singular software model that produces the optimal result for all possible conditions is highly improbable, this research hypothesizes that a GPU parallel processing model is advantageous in ARMAS systems configured to detect multiple faults for large quantity of system sensors. ARMAS systems containing a small quantity of system sensors are expected to see superior results with a CPU sequential processing model. Research results will quantify the amount of system sensors and faults associated with identifying the sequential to parallel processing efficiently cross-over point.

Furthermore, it proposes that a single ARMAS system instantiation is less computationally efficient than an association of smaller ARMAS systems working collectively to produce the same output under a set bounded conditions. The smaller ARMAS systems are formed with different subsets of sensors together forming the superset of sensors used in the single ARMAS system instantiation. The computational efficiency will be a measure of the overall execution time of an ARMAS program.

1.5 Assumptions/Limitations

The experiments accomplished in this study were performed on a Dell Precision 7720 workstation configured with a Intel Core I7-7920HQ CPU and an NIVIDA Quadro P5000 GPU. The Intel Core I7-7920HQ operates at 3.10 GHz with 8 MB of Cache. The workstation system total RAM contained 32 GB. The NIVIDA Quadro P5000 GPU is comprised of 2048 CUDA Cores operating at 1.51 GHz with 16 GB of GDDR5 memory. The optimization results presented in this thesis are particular to this hardware. However, the techniques in parallel computing and efficient software

model designs discussed in this thesis are valid optimization approaches for computer system with comparable hardware.

1.6 Thesis Contributions

The following are contributions made in this thesis:

1. The first C++ sequential processing implementation of ARMAS.
2. The earliest parallel processing implementation of ARMAS on a GPU device.
3. An identification of ARMAS system parameters where sequential to parallel processing efficiency cross-over points occur.
4. A novel ARMAS software model that:
    - uses layered ARMAS subsystems, and
    - requires less processing power than a single ARMAS system architecture instantiation.

1.7 Thesis Structure

This document is organized in the scholarly article thesis format. Chapter II provides a detailed explanation of relevant background information. This information is condensed into summarized versions in both scholarly articles contained in this thesis to support stand-alone publication. Chapter III is the first scholarly article that details the process of accelerating the ARMAS framework through parallel computing performed on a GPU device. Chapter IV is the second scholarly article that presents a novel software model that utilizes multiple ARMAS subsystems to generate a navigation solution with less computational demand. Finally, Chapter V discusses the conclusions drawn from the results.

II. Literature Review

This chapter presents the fundamental background information used to support the software design decisions in the subsequent chapters. Section 2.1 outlines the C++ programming language. Section 2.2 describes the heterogeneous computing model and parallel programming. Section 2.3 summarizes how the Kalman Filter uses state estimation to provide navigation solutions. Section 2.5 describes the Autonomous and Resilient Management of All-source Sensors (ARMAS) framework. Lastly, Section 2.6 discusses related research conducted.

2.1 C++ Programming Language

C++ is a general purpose programming language that supports objected-oriented programming and data abstraction [2]. It was first developed in 1979 in the Computing Science Research Center of Bell Labs in Murray Hill, New Jersery by Bjarne Stroustrup, with the first commercial release occurring in October 1985 [3][4]. C++ is now one of the most popular and widely used programming languages for system development [5][6]. There are an estimated 4.5 million C++ developers worldwide [7][8].

C++ is a superset of the C programming language (C11 standard), where most differences stem from C++'s greater emphasis on type checking. The greater emphasis on type checking allows a compiler to detect type errors, enabling programmers to easily correct typing mistakes or catch conceptual design flaws during early debugging. C++ is a complied language that supports low-level operations [3], which allows for direct interaction with hardware [5]. This provides a programmer with the capability to efficiently utilize available hardware resources for C++ programs. C++ is a flexible language that does not have inherent limitations that prevent particular

kinds of programs from being written, allowing for a large range of applications [2]. The versatility of the language allows it to be used in embedded systems, servers, investment banking, games, manufacturing, and even web applications [6][8][9].

The C++ programming language has a guiding principle that focuses on providing zero-overhead abstraction. This means a C++ program is not burdened by unused features of the language and it is optimized in such a way that eliminating the abstraction qualities of the code base will not increase performance [8][9]. Comparative studies have shown these defining characteristics enable C++ to achieve higher speed performance than other languages [10]. Additionally, C++ allows for the most memory-efficient program models with the least variability. Nevertheless, these advantages come at a cost as C++ programs tend to be two to three times as the length of scripting languages, such as Perl, Python, and Rexx [11]. Ultimately, the severity of this drawback can be offset by the programmer's ability to drive efficiency [12].

2.2 Heterogeneous Computing Model and Parallel Programming

In the early days, computers only contained a single Central Processing Unit (CPU), called a processing core, to run general computing tasks. The computing tasks were discrete series of calculations that executed in a successive order forming a sequential program that solved an overall problem. A CPU core is designed to handle very complex control logic that optimizes the execution of sequential programs [13]. A sequential program directly benefits from a CPU's ability to minimize the execution latency attained through short-latency cache memory accesses, low-latency arithmetic units, and sophisticated operand delivery logic [14].

Historically, this benefit allowed software developers to rely on hardware advances to increase the speed of their sequential programs. These early performance advances were achieved by continuously increasing the transistor count on a chip and the clock

rate frequency. This chip design improvement method slowed in 2003 due to energy consumption and heat dissipation issues [14][15]. Instead, researchers showed that better power efficiently could be achieved by having two cores running at lower frequencies than a single core operating at double the frequency of the two cores, while completing the same number of instructions in both instances [16][17]. This lead to a multi-core architecture, where multiple processing cores are on a single chip system [13].

The multi-core architecture eliminated the typical performance increase seen on new generations of CPUs with sequential programs as they were only able to run on one of the processing cores. Software developers adapted to these new architectures by implementing parallel programs instead. Parallel programs allow multiple threads to cooperate by performing calculations concurrently to complete a computing task faster [14][18]. Currently, parallelism is a driving force for computer architecture design. The multi-core architecture has now evolved to the manycore architecture, which describes chips with tens or hundreds of cores [13].

Graphical Processing Units (GPUs) represent a manycore architecture, which implements multiple levels of parallelism [17]. GPUs were originally only used for computing a large number of calculations required for displaying graphics [15]. The nature in which graphics computations are required led GPU chip designers to maximize the number of processing cores on a GPU chip at the expense of the cache memory size. The smaller cache memory on a GPU results in long latency memory accesses. In turn, the large number of processing cores increases the total execution throughput capability of the device[14].

In 2007, NVIDIA introduced the Compute Unified Device Architecture (CUDA), which opened up general purpose computing on GPUs (GPGPU). GPGPU allows for general computation for wide range of problems, not just those within the graphics

domain. GPGPU is ideal for problems involving large amounts of data parallelism [15]. Data parallelism emerges when there are many data elements that can be operated on independently. The large number of programmable cores within a GPU permit the independent operations to be done simultaneously. This has increased workload efficiency of parallel programs on data-parallel computation-intensive tasks [13].

The advantages of GPGPU has lead to a heterogeneous computing model that uses a suite of processor architectures to execute a computer program. A heterogeneous application seeks to run specific portions of a program on hardware components that computationally accelerate the execution of the overall application [13][19][20]. GPUs have become the most commonly used accelerator on computer systems [21], but other devices like Digital Signal Processors (DSPs) and Field-Programmable Gate Arrays (FPGAs) can also be used.

2.3 Kalman Filter in Navigation Applications

Navigation solutions based on accurate estimation of a vehicle's system states can be determined via model estimation. The Kalman Filter algorithm, developed by Rudolf E. Kalman in 1960 [22], is used in modern navigation systems to perform recursive model estimation for a process (dynamics) model in the form,

$$\dot{\mathbf{x}}(\mathbf{t}) \;=\; \mathbf{F}\mathbf{x}(\mathbf{t}) \;+\; \mathbf{B}\mathbf{u}(\mathbf{t}) \;+\; \mathbf{G}\mathbf{w}(\mathbf{t}), \tag{1}$$

where $\mathbf{x}$ is the system state vector, $\mathbf{u}$ is the system input control vector, and $\mathbf{w}$ is the white noise components vector. $\mathbf{F}$, $\mathbf{B}$, and $\mathbf{G}$ are linear operator matrices with constant coefficients for the state vector, control input vector, and noise vector, respectively. The discretization of the Kalman Filter equations using the Van Loan

method [23] allows the algorithm to be used in modern navigation systems to estimate a vehicle's system states.

Linear discrete sensor measurements modeled, is given by

$$\mathbf{z_k} \;=\; \mathbf{H x_k} \;+\; \mathbf{v_k}, \tag{2}$$

where $\mathbf{z}$ is the sensor measurement, $\mathbf{H}$ is the observation model that maps measurements to the states, and $\mathbf{v}$ is the white noise function, are used to update the state estimate at timepoint k. The discrete Kalman Filter algorithm estimates system states by propagating an initial state estimate using

$$\hat{\mathbf{x}}^{-}_{\mathbf{k+1}} = \; \Phi \hat{\mathbf{x}}^{+}_{\mathbf{k}} \;+\; \mathbf{B_d u_k}, \tag{3}$$

$$\mathbf{P}^{-}_{\mathbf{k+1}} = \; \Phi \mathbf{P}^{+}_{\mathbf{k}} \Phi^{\mathbf{T}} \;+\; \mathbf{Q_d}, \tag{4}$$

where $\mathbf{x}$ is the state estimates, Φ is the discrete state transition, and $\mathbf{P}$ is the associated state error covariance matrix. The states and covariance are updated by combining propagated state estimates and measurement readings via the Kalman gain, $\mathbf{K}$, using

$$\mathbf{K_k} = \; \mathbf{P}^{-}_{\mathbf{k}} \mathbf{H}^{\mathbf{T}} \left[\mathbf{H P_k H}^{\mathbf{T}} + \mathbf{R} \right]^{-\mathbf{1}}, \tag{5}$$

$$\hat{\mathbf{x}}^{+}_{\mathbf{k}} = \hat{\mathbf{x}}^{-}_{\mathbf{k}} \; + \mathbf{K_k} \left[\mathbf{z_k} - \; \mathbf{H} \hat{\mathbf{x}}^{-}_{\mathbf{k}} \right], \tag{6}$$

$$\mathbf{P}^{+}_{\mathbf{k}} = (\mathbf{1} \; - \; \mathbf{K_k H}) \mathbf{P}^{-}_{\mathbf{k}}, \tag{7}$$

where $\mathbf{R}$ is the measurement error covariances.

In summary, given a correctly-modeled state transition and measurement system representations, an optimal stochastic estimation of a vehicle's system states can be determined by recursively implementing the Kalman Filter algorithm. A single it-

eration of the Kalman Filter algorithm is performed in two phases: propagate and update. The propagate phase computes Equations 3 and 4. The update phase executes Equations 5 through 7 [24].

2.4 Extended Kalman Filter

Navigation systems that use non-linear dynamics or measurement models may not always achieve optimal results from implementing the linear Kalman filter equations. In certain cases, the non-linear systems can be transformed in linear approximations to produce accurate solutions. In these cases, the non-linear systems employ the Extended Kalman filter (EKF) [25][26] which uses the non-linear system dynamics equation in the form,

$$\dot{\mathbf{x}}(\mathbf{t}) = \mathbf{f}[\mathbf{x}(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}] + \mathbf{G}(\mathbf{t})\mathbf{w}(\mathbf{t}), \tag{8}$$

where $\mathbf{f}$ is a vector that contains functions that represent the system. The non-linear measurement model is given by,

$$\mathbf{z_k} = \mathbf{h}[\mathbf{x_k}, \mathbf{t_k}] + \mathbf{v_k}, \tag{9}$$

where $\mathbf{h}$ is a vector of functions that models the system sensor. To linearize the non-linear system, the states are converted using the perturbation model given by,

$$\delta\mathbf{x}(\mathbf{t}) \triangleq \mathbf{x}(\mathbf{t}) - \hat{\mathbf{x}}(\mathbf{t}), \tag{10}$$

where $\delta\mathbf{x}(\mathbf{t})$ is the difference between the state estimate and the true state vector. To propagate the system to the time of the next measurement, the EKF integrates the

non-linear dynamics equation over the time difference using

$$\hat{\mathbf{x}}_{\mathbf{k+1}}^{-} = \int_{\mathbf{t_k}}^{\mathbf{t_{k+1}}} \mathbf{f}[\mathbf{x}(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}]\mathbf{dt} + \hat{\mathbf{x}}_{\mathbf{k}}^{+}. \tag{11}$$

The state covariance propagation does not change and continues using Equation (7), while the linearized dynamics model matrix is described by

$$\mathbf{F_k} = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\hat{x}_k^+}. \tag{12}$$

To update the state estimates using sensor measurement which may be non-linear, the measurement is initially predicted by evaluating the measurement model equation with the most recent estimate using

$$\hat{\mathbf{z}}_{\mathbf{k}} = \mathbf{h}[\hat{\mathbf{x}}_{\mathbf{k}}^{-}, \mathbf{t}_{\mathbf{k}}], \tag{13}$$

$$\delta \mathbf{z}_{\mathbf{k}} = \mathbf{z}_{\mathbf{k}} - \hat{\mathbf{z}}_{\mathbf{k}}, \tag{14}$$

where $\delta \mathbf{z_k}$ is named the pre-update measurement residual which describes the difference between the actual and predicted measurements. By linearizing the non-linear measurement function $\mathbf{h}$ using

$$\mathbf{H_k} = \left.\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right|_{\hat{x}_k^-} \tag{15}$$

where H is the linearized matrix, the propagated and measured state estimates can be combined by calculating the Kalman gain using Equation (5), reducing the measurement update equation to

$$\delta \hat{\mathbf{x}}_{\mathbf{k}}^{+} = \mathbf{K_k} \delta \mathbf{z_k}, \tag{16}$$

by utilizing perturbation state estimates and measurements.

The perturbation state $\delta \hat{\mathbf{x}}$, starts at zero during each filter iteration and is updated

using Equation (16). It is then added to the nominal trajectory to produce a nominal estimate. The perturbation state is then reset to zero after each filter iteration.

2.5 Autonomous Resilient Management of All-source Sensors (ARMAS)

Developed in 2018, ARMAS is a generalized framework for real-time management of heterogeneous, asynchronous all-source sensors [1]. The framework identifies corrupt signals from mismodeled, uncalibrated, and/or faulty sensors through sensor validation, Fault Detection and Exclusion (FDE), recalibration, and remodeling modes via single software package. ARMAS utilizes a set of SCORPION [27] pluggable EKF estimators that provide a navigation solution to problems in the nonlinear form,

$$\dot{\mathbf{x}}(\mathbf{t}) = \mathbf{f}[\mathbf{x}(\mathbf{t}), \epsilon(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}] + \mathbf{G}(\mathbf{t})\mathbf{w}(\mathbf{t}), \tag{17}$$

where $\mathbf{x}$ is a $N \times 1$ state vector of a vehicle's position, velocity, and attitude. The measurement error states vector ϵ is of dimension $M \times 1$, $\mathbf{u}$ is the control input vector, $\mathbf{G}$ is an $(N + M) \times W$ linear operator, and $\mathbf{w}$ is a $W \times 1$ white noise process described by a $W \times W$ continuous process noise strength matrix, $\mathbf{Q}$.

The state estimates are propagated by using the non-linear state dynamics model and measurements updates from $j = 1...J$ available sensors. The measurement model for the j^{th} sensor is characterized by

$$\mathbf{z_k^{[j]}} = \mathbf{h^{[j]}}[\mathbf{x}(\mathbf{t}), \epsilon^{[\mathbf{j}]}(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}, \mathbf{p^{[j]}}] + \mathbf{v_k^{[j]}}, \tag{18}$$

where $\mathbf{h^{[j]}}$ is the nonlinear measurement function for the j^{th} sensor, $\epsilon^{[\mathbf{j}]}$ is an L x 1 subset of ϵ that contains additional error states required to process sensor measurements, $\mathbf{p^{[j]}}$ is a $P \times 1$ user-selectable model parameter vector for $\mathbf{h^{[j]}}$, and $\mathbf{v_k^{[}j]}$ is a $Z \times 1$ discrete white noise process with covariance defined by matrix $\mathbf{R_k^{[j]}}$.

The $Z \times 1$ measurement Kalman Filter residual, $\mathbf{r_k^{[j]}}$, for sensor j, is defined by

$$\mathbf{r_k^{[j]}} = \mathbf{z_k^{[j]}} - \mathbf{h^{[j]}}[\hat{\mathbf{x}}_\mathbf{k}^-, \hat{\epsilon}_\mathbf{k}^{[\mathbf{j}]-}, \mathbf{u_k}, \mathbf{t_k}, \hat{\mathbf{p}}_\mathbf{k}^{[\mathbf{j}]}], \tag{19}$$

where $\hat{\mathbf{x}}_\mathbf{k}^-$, $\hat{\epsilon}_\mathbf{k}^{[\mathbf{j}]-}$, and $\hat{\mathbf{p}}_\mathbf{k}^{[\mathbf{j}]}$ are estimated quantities. Assuming white Gaussian noise, the residual vector from Equation 19 is expected to follow the distribution

$$\mathbf{r_k^{[j]}} \hookrightarrow \mathcal{N}(\mathbf{0_{Nx1}}, \mathbf{S_k^{[j]}}), \tag{20}$$

$$\mathbf{S_k^{[j]}} = \mathbf{H_k^{[j]}} \mathbf{P_k^-} \mathbf{H_k^{[j]^T}} + \mathbf{R_k^{[j]}}, \tag{21}$$

where $\mathbf{P_k^-}$ is the $(N+M) \times (N+M)$ state estimate error covariance matrix at time t_k and $\mathbf{H_k^{[j]^T}}$ is the $Z \times (N+M)$ Jacobian of $\mathbf{h^{[j]}}$.

Sensors are initialized as trusted or untrusted. Untrusted sensors are placed in sensor validation mode before they are accepted into monitoring mode. The validation mode in ARMAS utilizes the likehood function

$$\mathcal{L}_{\mathbf{N_k}} = \sum_{\mathbf{i=k-N+1}}^{\mathbf{k}} -\log(|\mathbf{2}\pi\mathbf{\Sigma_i}|) - \frac{\mathbf{1}}{\mathbf{2}}\mathbf{r_i^T}\mathbf{\Sigma_i^{-1}}\mathbf{r_i}, \tag{22}$$

where

$$\mathbf{\Sigma}_i = \mathbf{H}\mathbf{P}_i^-\mathbf{H}^T + \mathbf{R} \tag{23}$$

and the $\mathbf{N}$ is the most recent residuals, to monitor the statistical distribution of a user-defined monitoring period comprised of recent Kalman pre-update residuals. A Chi-squared, $\mathcal{X}^*$, statistic test is used to detect abberations outside a user-defined threshold over a specified sampling duration. Sensors in validation mode are excluded from the main state estimate results.

Conversely, trusted sensors are immediately brought online into monitoring mode.

In monitoring mode, sensor measurements are directly used to update the main state estimates. The same pre-update residual likelihood function used in the validation mode is utilized to monitor sensor performance. The ARMAS monitor mode will be discussed in-depth in the section 2.5.1.

If an abnormality is detected during sensor performance monitoring, the sensor becomes "untrusted" and is quarantined from affecting the core navigation state estimates solution, $\hat{\mathbf{x}}^{[\mathbf{j}]}$. ARMAS will attempt to reinitialize the untrusted sensor through validation mode. If this fails, ARMAS will then try to repair and recover the untrusted sensor via two separate modes: sensor calibration and remodeling.

In calibration mode, user-selectable sensor parameters, $\mathbf{p}^{[\mathbf{j}]}$, and/or $\epsilon^{[\mathbf{j}]}$ are re-estimated using residual monitoring from trusted sensors that have observability of the core navigation state estimates solution. When a single calibration parameter exist, ARMAS tries to correct the calibration with residual monitoring and moves the sensor back to validation mode. When linked extrinsic calibration parameters exist (e.g. camera lever arm and camera orientation within $\mathbf{p}^{[\mathbf{j}]}$ or $\epsilon^{[\mathbf{j}]}$), each calibration parameter is estimated individually and sequenced based on convergence of the state covariance to maintain state observability.

If the calibration mode fails to bring the sensor back online, the sensor is moved to the remodeling mode where ARMAS attempts to alter the measurement model, $\mathbf{h}^{[\mathbf{j}]}$, based on 1...$S$ user-defined measurement models. S simultaneous filters, each with a unique measurement model, are generated and a specified period of measurement residuals is compared against the core navigation state estimates solution. The filter that best matches the prescribed distribution described by Equation (20) during the specified period is selected as the new sensor measurement model and the sensor is moved to validation mode. If the remodeling mode fails to select a new sensor measurement model and the Resilient Sensor Recovery (RSR) option in ARMAS is

active, the sensor re-enters validation mode after a user-selectable time frame in an effort phase out temporal anomalies. Figure 1 shows the state transition diagram of the discussed modes. The result is a software framework that supports heterogeneous, asynchronous all-source sensors resilient against various sensor calibration, modeling, and temporal faults.

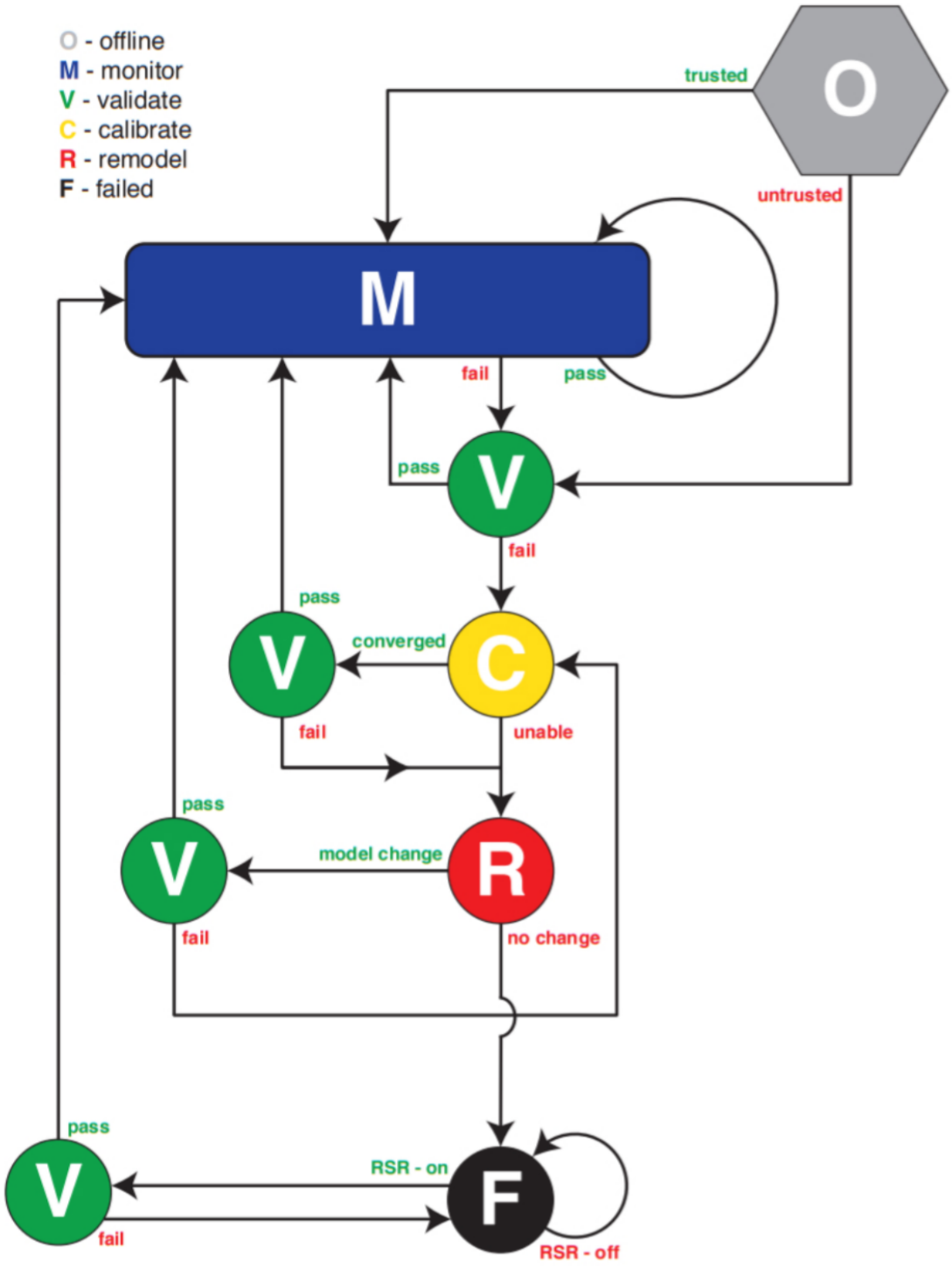

Figure 1: ARMAS Framework State Diagram [1]

2.5.1 Monitoring Mode: Sensor-Agnostic All-source Residual Monitoring (SAARM)

The ARMAS monitoring mode employs SAARM to detect multiple sensor failure modes, such as, bias, mismatched model, and/or miscalibration. To detect a single sensor failure, $J = I$ differently configured navigation subfilters are maintained for comparison. This configuration is subsequently extended to provide resiliency to multiple simultaneous faults with the utilization of more processing power.

SAARM operates on a system of the form

$$\dot{\mathbf{x}}(\mathbf{t}) = \mathbf{f}[\mathbf{x}(\mathbf{t}), \epsilon(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}] + \mathbf{G}(\mathbf{t})\mathbf{w}(\mathbf{t}), \tag{24}$$

where $\mathbf{x}$ is a $N \times 1$ state vector of a vehicle's position, velocity, and attitude. The measurement error states vector ϵ is of dimension $M \times 1$, $\mathbf{u}$ is the control input vector, $\mathbf{G}$ is an $(N + M) \times W$ linear operator, and $\mathbf{w}$ is a $W \times 1$ white noise process described by a $W \times W$ continuous process noise strength matrix, $\mathbf{Q}$.

SAARM estimates system states using J individual subfilters. At time $t = t_k$, the system state vector and state estimate covariance matrix are characterized by

$$\hat{\mathbf{x}}^{[\mathbf{j}]}(\mathbf{t_k}) \textbf{ and } \mathbf{P}^{[\mathbf{j}]}_{\hat{\mathbf{x}}\hat{\mathbf{x}}}(\mathbf{t_k}) \textbf{ for j } = \textbf{ 1...J individual subfilters}. \tag{25}$$

Each of the individual subfilters incorporates a subset of I-1 sensors. At time $t = t_k$, the i^{th} sensor supplies measurements defined by

$$\mathbf{z}^{[\mathbf{i}]}(\mathbf{t_k}) = \mathbf{h}^{[\mathbf{i}]}[\hat{\mathbf{x}}^{[\mathbf{j}]}(\mathbf{t_k^-}), \mathbf{u}(\mathbf{t_k}), \mathbf{t_k}] + \mathbf{v}^{[\mathbf{i}]}(\mathbf{t_k}), \tag{26}$$

where $\mathbf{h}^{[\mathbf{i}]}$ is the nonlinear measurement function, $\mathbf{u}(\mathbf{t_k})$ is the control input function, and $\mathbf{v}^{[\mathbf{i}]}(\mathbf{t_k})$ is a $Z_i \times 1$ discrete white noise process with covariance defined by matrix

$\mathbf{R}^{[\mathbf{i}]}(\mathbf{t_k})$.

The pre-update measurement estimate for sensor i from filter j is given by

$$\hat{\mathbf{z}}^{[\mathbf{i,j}]}(\mathbf{t_k^-}) = \mathbf{h}^{[\mathbf{i}]}[\hat{\mathbf{x}}^{[\mathbf{j}]}(\mathbf{t_k^-}), \mathbf{u}(\mathbf{t_k}), \mathbf{t_k}], \tag{27}$$

where the estimated covariance matrix is described by

$$\mathbf{P}^{[\mathbf{i,j}]}_{\hat{\mathbf{z}}\hat{\mathbf{z}}}(\mathbf{t_k^-}) = \mathbf{H}^{[\mathbf{i}]}(\mathbf{t_K^-})\mathbf{P}_{\hat{\mathbf{x}}\hat{\mathbf{x}}}(\mathbf{t_K^-})\mathbf{H}^{[\mathbf{i}]}(\mathbf{t_K^-})^{\mathbf{T}}. \tag{28}$$

Using Equations (27) and (28), the "pre-update residual" vector between sensor i and filter j, $\mathbf{r}^{[\mathbf{i,j}]}$ and its associated covariance matrix, $\mathbf{P}^{[\mathbf{i,j}]}_{\mathbf{rr}}$ are generated using

$$\mathbf{r}^{[\mathbf{i,j}]}(\mathbf{t_k}) = \mathbf{z}^{[\mathbf{i}]}(\mathbf{t_k}) - \hat{\mathbf{z}}^{[\mathbf{i,j}]}(\mathbf{t_k^-}), \tag{29}$$

$$\mathbf{P}^{[\mathbf{i,j}]}_{\mathbf{rr}}(\mathbf{t_k}) = \mathbf{R}^{[\mathbf{i}]}(\mathbf{t_k}) + \mathbf{P}^{[\mathbf{i,j}]}_{\hat{\mathbf{z}}\hat{\mathbf{z}}}(\mathbf{t_k^-}). \tag{30}$$

ARMAS fault detection involves calculating a moving average of recent residual-space test statistics formed by pre-update residual vectors from Equations (29) and (30). This calculation enables ARMAS to detect three types of faults: a bias, an incorrectly stated noise covariance, or an incorrectly stated measurement model. The likelihood function uses a single residual-space statistic which determines if a set of observed residuals between a specific sensor-filter pair are adhering to their expected distribution. The single residual-space statistic is based on the Mahalanobis distance, $\mathbf{d}$, defined by

$$\mathbf{d^2} = (\mathbf{y} - \mu)^{\mathbf{T}}\mathbf{\Sigma^{-1}}(\mathbf{y} - \mu), \tag{31}$$

where μ is the mean and $\mathbf{\Sigma}$ is the covariance of a Z_i-dimensional Gaussian distribution. Given that a sum of M independent $\mathbf{d^2}$ distances follows a $\mathcal{X}^*$ distribution with Z

degrees of freedom [28] shown by

$$\mathcal{X}^* = \sum_{\mathbf{s=k}}^{\mathbf{k+M}} \mathbf{d^2(t_s)}, \tag{32}$$

$$\mathbf{d^2(t_k)} = \mathbf{r^T(t_k)[P_{rr}(t_k)]^{-1}r(t_k)}, \tag{33}$$

the set of pre-update residuals is known to be a zero-mean, white sequence [25]. The fault detection test for M pre-residuals is composed of the following hypotheses:

$$\mathbf{H_0} : \mathcal{X}^*_{\mathbf{[i,j]}} < \mathcal{X}^{\mathbf{2}}(\mathbf{1} - \alpha/\mathbf{2}, \mathbf{M\ x\ Z_i}), \tag{34}$$

$$\mathbf{H_1} : \mathcal{X}^*_{\mathbf{[i,j]}} > \mathcal{X}^{\mathbf{2}}(\mathbf{1} - \alpha/\mathbf{2}, \mathbf{M\ x\ Z_i}), \tag{35}$$

where α is the probability of false alarm and M is the number of averaged pre-residual samples. $\mathbf{H_0}$ is the hypothesis where the fault is not present in filter j. $\mathbf{H_1}$ is the hypothesis where a fault is present in filter j. The resulting hypothesis test forms the foundational principle behind the fault detection algorithm.

Once a fault is detected, a consensus of multiple subfilters is utilized to distinguish the faulty sensor. With $J = I$ subfilters, SAARM can only exclude single faults within each residual monitoring time frame (i.e. M-sample moving average). In this case, each subfilter is comprised by a different subset of $I-1$ sensors, therefore each subfilter is missing a unique single sensor. This assumes that all states are observable by all subfilters. In addition to $J = I$ subfilters, a main filter is maintained in ARMAS to generate a full navigation solution for user output. Accordingly, cross-covariance terms between the main filter and any other filters are not used for any computation. For this case, SAARM guarantees, under the assumption that, at most, one sensor can fail simultaneously, at least one of the J subfilters will be completely unaffected by faulty measurements [1].

The fault identification procedure registers a T-matrix of dimension I x J using

$$\mathbf{T(i,j)} = \begin{cases} \mathbf{0}, \text{Sensor i not associated with filter j} \\ \mathbf{0}, \mathcal{X}^{*}_{[\mathbf{i,j}]} < \mathcal{X}^{\mathbf{2}}(\mathbf{1}-\alpha/\mathbf{2}, \mathbf{M\ x\ Z_i}), \text{No Fault Detected, } \mathbf{H_0} \\ \mathbf{1}, \mathcal{X}^{*}_{[\mathbf{i,j}]} > \mathcal{X}^{\mathbf{2}}(\mathbf{1}-\alpha/\mathbf{2}, \mathbf{M\ x\ Z_i}), \text{Fault Detected, } \mathbf{H_1} \end{cases} \tag{36}$$

Figure 2 depicts the association of I sensors and J subfilters used to establish the consensus of subfilters for faulty sensor identification. The rows correspond to the $i = 1...I$ sensors, while the columns coincide to the $j = 1...J$ subfilters. Each row incorporates measurements, $\mathbf{Z}^{[\mathbf{i}]}$, and measurement error covariances, $\mathbf{R}^{[\mathbf{i}]}$ from the i^{th} sensor. Each column incorporates the estimated measurements, $\hat{\mathbf{z}}^{[\mathbf{i,j}]}$, and its associated error covariances, $\mathbf{P}^{[\mathbf{i,j}]}_{\hat{\mathbf{z}}\hat{\mathbf{z}}}$.

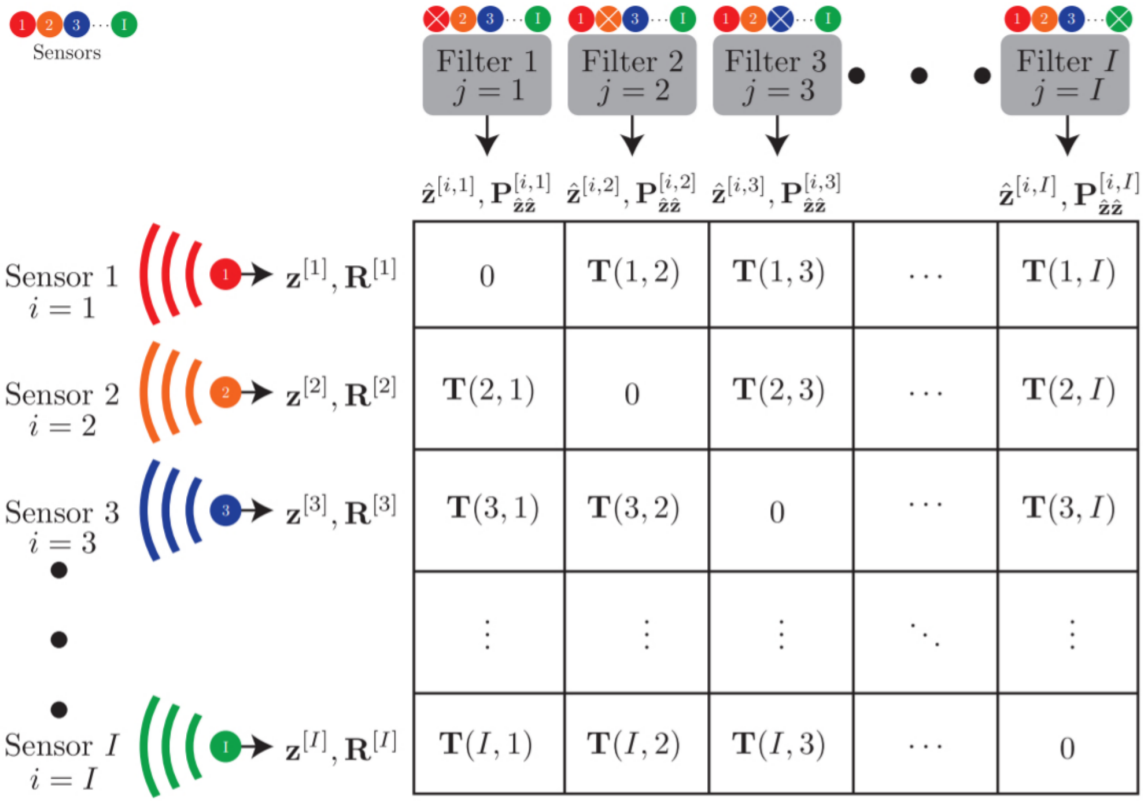

Figure 2: SAARM T-Matrix for i = 1...I Sensors [1]

Based on this procedure, a fault is declared when T contains any non-zero entries. This means at least one subfilter detected $\mathbf{H_1}$. Once a fault is declared, the remaining subfilters reach a consensus that leaves a single subfilter fault-free. This faultly sensor is identified by determining the sensor not included in the fault-free subfilter. This elevates the fault-free subfilter to "main filter" status. The new main filter distributes the state estimate and covariance to the rest of the subfilter in order to re-initialize them. This effectively eliminates the faulty sensor from the navigation solution as each subfilter now contains $I-2$ sensors. The faulty sensor is then moved to validation mode following the state diagram transition shown in Figure 1. The procedure can be repeated to detect multiple serial faults under the assumption that the successive faults do not occur during the first M samples after re-initializing the subfilters.

In order for ARMAS to detect and manage multiple simultaneous faults, multiple layers of subfilters must be maintained. The number of concurrent subfilters, J_N, required to handle N simultaneous faults for I sensors is calculated using

$$\mathbf{J_N} = \binom{\mathbf{I}}{\mathbf{I-N}} = \frac{\mathbf{I!}}{\mathbf{N!(I-N)!}}. \tag{37}$$

Given the formula above, one can see that the computing power required as sensors and faults are scaled up becomes significant very quickly. This research seeks to address this computation requirement by efficiently distributing the computational load across available hardware resources in an effort to maximize the processing capability.

2.6 Related Work

Although the Kalman Filter algorithm can be traced back to 1960 [22], it is only until recently that the acceleration of the algorithm on computer systems has been explored. In 2011, Min-Yu Huang *et al.* [29] showed how the algorithm could

be accelerated by parallelizing the Kalman Filter matrix computations on a GPU. The study decomposed the algorithm into the individual math operations required for a full iteration and analyzed the clock cycles required for each operation. The matrix inversion operation was shown to be the most time-consuming, followed by matrix multiplication. Transferring these types of calculations to the GPU decreased computation times as the data dimensions increased from 1000 to 7000. Final results demonstrated a GPU implementation provided significant improvement over a CPU, achieving a maximum speedup of 7398x when the observation dimension was 6000 and the state dimension was 4500. This approach has been utilized on multiple different applications [30][31][32][33][34][35].

An alternate method was present by Rosen in [36] and [37], which breaks data dependencies by reordering the Kalman Filter equations. This allows for an almost completely parallel algorithm implementation, however it is only applicable to specialized Kalman filter applications. Thein and Kaung showed a similar approach [38] could be used for general Kalman Filter application by computing decoupled equations on separate processors decreasing overall execution time of the algorithm.

Karimipour and Dinavahi presented a systematic process in [39] that also considered the application employing the Kalman Filter algorithm to maximize the extraction of task and data parallelism. These portions of the software were then executed on a multi-core processor to accelerate the system. Results showed a total speed up of 15x over a sequential application.

Lastly, [40] and [41] are the studies most comparable to this research. Both studies implemented multiple Kalman Filter instances on a GPU. Results in [40] showed a 20 to 30 percent performance increase over a CPU. However, the study conducted by Yussiff *et al.* [40] disclosed few implementation details which in turn had minimal contribution to this research work. On the other hand, the study done by Amamra

and Aouf [41] detailed the GPU kernel code that was utilized, the CPU to GPU data transfer method, and the data layout in memory. This information bolstered the software design decisions in the ensuing chapters.

In conclusion, this research is unique in that it builds off Dr. Juan Jurado's development of the ARMAS framework [1], which uses Kalman Filters to provide a navigation solution resilient to sensor failures. For this research work, a sensor can be any entity that provides navigation readings including a GPS satellite. Detecting sensor failures requires a bank of multiple Kalman Filter instances running in the background in addition to a main Kalman Filter which is responsible for providing the navigation solution to the user. When a sensor failure occurs, the main Kalman Filter is replaced by the appropriate Kalman Filter in the bank to exclude the failed sensor from the navigation solution. This requires additional logic and control routines that have not been implemented in Kalman Filter optimization studies discussed in this section.

III. Scholarly Article: Optimizing a Bank of Kalman Filters for Navigation Integrity using Parallel Computing

Abstract

Alternative navigation is an area of research that employs a variety of sensor technologies to provide a navigation solution in Global Navigation Satellite System degraded or denied environments. The Autonomy and Navigation Technology Center at the Air Force Institute of Technology has recently developed the Autonomous and Resilient Management of All-source Sensors (ARMAS) navigation framework which utilizes an array of Kalman filters to provide a navigation solution resilient to sensor failures. The Kalman filter array size increases exponentially as system sensors and detectable faults are scaled up, which in turn increases the computational power required to run ARMAS in a real-world application. In an effort to engineer a real-time ARMAS system, this study developed C++ CPU and GPU versions to examine the performance trade-offs as system sensors and detectable faults are scaled up. Small sensor size configurations proved to be faster on a CPU, while a 3.5 order of magnitude speedup was achieved by the GPU ARMAS implementation over the CPU version on a large sensor system configuration. Results show promise that a real-time ARMAS system can be achieved for large scale applications through parallel processing on a many-core processor architecture.

3.1 Introduction

Alternative navigation is an area of research which employs a variety of sensor technologies to provide a navigation solution in Global Navigation Satellite System (GNSS) degraded or denied environments [42]. The Autonomy and Navigation Technology (ANT) Center at the Air Force Institute of Technology (AFIT) has recently

developed the Autonomous and Resilient Management of All-source Sensors (ARMAS) navigation framework which produces a navigation solution resilient to sensor failures [1]. ARMAS employs a sensor fault detection and exclusion procedure that utilizes a bank of overlapping, but unique, Kalman filters which identify faulty sensors through a general consensus. One key benefit of this procedure is that an uncorrupted Kalman filter always exists within an ARMAS system properly configured to handle sensor fault case.

To transition the academic software to an operational real-time system, the non-trivial computational power required to run ARMAS must be addressed. In specific, ARMAS utilizes an array of unique Kalman filters which identify faulty sensors through a general consensus. The size of the Kalman filter array depends on the number of system sensors and detectable system sensor faults specified by the user. As a general approximation, the Kalman filter array size increases exponentially as system sensors and detectable faults are scaled up linearly, which in turn increases the computational load required to run ARMAS in a real-world application.

In an effort to engineer an operational real-time ARMAS system, this study developed sequential and parallelized versions of ARMAS to examine the performance trade-offs as system sensors and detectable faults are scaled up. This research aims to identify advantageous software design specifications for a real-time ARMAS system.

3.2 Background

3.2.1 Kalman Filter

The Kalman filter, developed by Rudolf E. Kalman in 1960 [22], is used to perform recursive state estimation for a continuous-time process model in the form,

$$\dot{\mathbf{x}}(\mathbf{t}) \;=\; \mathbf{F}\mathbf{x}(\mathbf{t}) \;+\; \mathbf{B}\mathbf{u}(\mathbf{t}) \;+\; \mathbf{G}\mathbf{w}(\mathbf{t}), \tag{38}$$

where $\dot{\mathbf{x}}$ is the system state vector, $\mathbf{u}$ is the system input control vector, and $\mathbf{w}$ is the white noise components vector. $\mathbf{F}$, $\mathbf{B}$, and $\mathbf{G}$ are linear operator matrices with constant coefficients for the state vector, control input vector, and noise vector, respectively. The discretization of the Kalman filter equations using the Van Loan method [23], allows the algorithm to be used in modern navigation systems to estimate a vehicle's system states.

Linear discrete sensor measurements modeled by

$$\mathbf{z_k} \;=\; \mathbf{H x_k} \;+\; \mathbf{v_k}, \tag{39}$$

where, $\mathbf{z}$ is the sensor measurement, $\mathbf{H}$ is the observation model which maps measurements to the states, and $\mathbf{v}$ is the white noise function, are used to update the state estimate at timepoint $\mathbf{k}$.

The Kalman filter algorithm is performed by iterating over 2 phases: propagate and update. The Kalman filter propagate phase is the numerical integration of Equation (38) for a state state estimate and state error covariance from timepoint $\mathbf{k}$ to $\mathbf{k+1}$, using

$$\hat{\mathbf{x}}^{-}_{\mathbf{k+1}} = \; \Phi \hat{\mathbf{x}}^{+}_{\mathbf{k}} \;+\; \mathbf{B_d u_k}, \tag{40}$$

$$\mathbf{P}^{-}_{\mathbf{k+1}} = \; \Phi \mathbf{P}^{+}_{\mathbf{k}} \Phi^{\mathbf{T}} \;+\; \mathbf{Q_d}, \tag{41}$$

where $\mathbf{x}$ is the state estimates, Φ is the discrete state transition, and $\mathbf{P}$ is the associated state error covariance matrix. The states and covariance are updated by comparing actual measurements with estimated measurements modeled by Equation (39) and using the Kalman gain, $\mathbf{K}$, using

$$\mathbf{K_k} = \; \mathbf{P^{-}_k H^T} \left[\mathbf{H P_k H^T + R} \right]^{-\mathbf{1}}, \tag{42}$$

$$\hat{\mathbf{x}}_{\mathbf{k}}^{+} = \hat{\mathbf{x}}_{\mathbf{k}}^{-} + \mathbf{K_k}\left[\mathbf{z_k} - \mathbf{H}\hat{\mathbf{x}}_{\mathbf{k}}^{-}\right], \tag{43}$$

$$\mathbf{P}_{\mathbf{k}}^{+} = (\mathbf{1} - \mathbf{K_k H})\mathbf{P}_{\mathbf{k}}^{-}, \tag{44}$$

where, $\mathbf{R}$ is the measurement error covariances.

3.2.2 Extended Kalman Filter

Systems that use non-linear dynamics or measurement models may not always achieve optimal results from implementing the linear Kalman filter equations. In certain cases, the non-linear systems can be transformed in linear approximations to produce accurate solutions. In these cases, the non-linear systems employ the Extended Kalman filter (EKF) [25][26] which uses the non-linear system dynamics equation in the form,

$$\dot{\mathbf{x}}(\mathbf{t}) = \mathbf{f}[\mathbf{x}(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}] + \mathbf{G}(\mathbf{t})\mathbf{w}(\mathbf{t}), \tag{45}$$

where $\mathbf{f}$ is a vector that contains functions that represent the system. The non-linear measurement model is given by,

$$\mathbf{z_k} = \mathbf{h}[\mathbf{x_k}, \mathbf{t_k}] + \mathbf{v_k}, \tag{46}$$

where $\mathbf{h}$ is a vector of functions that models the system sensor. To linearize the non-linear system, the states are converted using the perturbation model given by,

$$\delta\mathbf{x}(\mathbf{t}) \triangleq \mathbf{x}(\mathbf{t}) - \hat{\mathbf{x}}(\mathbf{t}), \tag{47}$$

where $\delta\mathbf{x}(\mathbf{t})$ is the difference between the state estimate and the true state vector. To propagate the system to the time of the next measurement, the EKF integrates the

non-linear dynamics equation over the time difference using

$$\hat{\mathbf{x}}_{\mathbf{k+1}}^{-} = \int_{\mathbf{t_k}}^{\mathbf{t_{k+1}}} \mathbf{f}[\mathbf{x}(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}]\mathbf{dt} + \hat{\mathbf{x}}_{\mathbf{k}}^{+}. \tag{48}$$

The state covariance propagation does not change and continues using Equation (44), while the linearized dynamics model matrix is described by

$$\mathbf{F_k} = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\hat{x}_k^+}. \tag{49}$$

To update the state estimates using sensor measurement which may be non-linear, the measurement is initially predicted by evaluating the measurement model equation with the most recent estimate using

$$\hat{\mathbf{z}}_{\mathbf{k}} = \mathbf{h}[\hat{\mathbf{x}}_{\mathbf{k}}^{-}, \mathbf{t_k}], \tag{50}$$

$$\delta \mathbf{z_k} = \mathbf{z_k} - \hat{\mathbf{z}}_{\mathbf{k}}, \tag{51}$$

where $\delta \mathbf{z_k}$ is named the pre-update measurement residual which describes the difference between the actual and predicted measurements. By linearizing the non-linear measurement function $\mathbf{h}$ using

$$\mathbf{H_k} = \left.\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right|_{\hat{x}_k^-} \tag{52}$$

where H is the linearized matrix, the propagated and measured state estimates can be combined by calculating the Kalman gain using Equation (42), reducing the measurement update equation to

$$\delta \hat{\mathbf{x}}_{\mathbf{k}}^{+} = \mathbf{K_k} \delta \mathbf{z_k}, \tag{53}$$

by utilizing perturbation state estimates and measurements.

The perturbation state $\delta \hat{\mathbf{x}}$, starts at zero during each filter iteration and is updated

using Equation (53). It is then added to the nominal trajectory to produce a nominal estimate. The perturbation state is then reset to zero after each filter iteration.

3.2.3 ARMAS

ARMAS is a software framework that provides autonomous sensor management for navigation systems [1]. The ARMAS framework for this study is primarily designed around the Extended Kalman Filter but can be adapted to other similar filtering techniques, such as the traditional Kalman filter or unscented Kalman filter. The state estimates are propagated by using the state dynamics process model and updated using measurements from $j = 1...J$ available sensors as described in Equations (40)-(44).

ARMAS provides sensor failure resiliency through statistical residual monitoring based on a Mahalanobis distance likelihood function. This approach requires ARMAS to maintain an array of uniquely configured Kalman filters to detect sensor failures in addition to a main filter that produces the navigation solution for the user. To detect sensor failure(s) in a system, a set of J uniquely configured Kalman filters obtain a consensus output that identifies the faulty sensor(s). To detect a single sensor fault, $J = I$ (number of system sensors). In this situation, each Kalman filter contains a unique subset of $I - 1$ sensors, thus allowing a different sensor to be excluded in each filter. For this condition, under the assumption that at most one sensor can fail for this case, at least one of the J Kalman filters will be completely unaffected by the faulty measurement. The faulty sensor procedure populates a test results matrix, T, with

$$\mathbf{T(i,j)} = \begin{cases} \mathbf{0}, \text{Sensor i not associated with filter j} \\ \mathbf{0}, \text{No Fault Detected} \\ \mathbf{1}, \text{Fault Detected} \end{cases} \tag{54}$$

as illustrated in Figure 3.

In order for ARMAS to detect and manage multiple simultaneous faults, multiple layers of subfilters must be maintained. The number of concurrent subfilters, J_N, required to handle N simultaneous faults for I sensors is calculated using

$$\mathbf{J_N} = \binom{\mathbf{I}}{\mathbf{I}-\mathbf{N}} = \frac{\mathbf{I}!}{\mathbf{N}!(\mathbf{I}-\mathbf{N})!}. \tag{55}$$

Given the I choose $(I-N)$ formula above, one can calculate that the computing power required as sensors and faults are scaled up becomes significant very quickly. This research seeks to address this computation requirement by efficiently distributing the computational load across available hardware resources in an effort to maximize the processing capability.

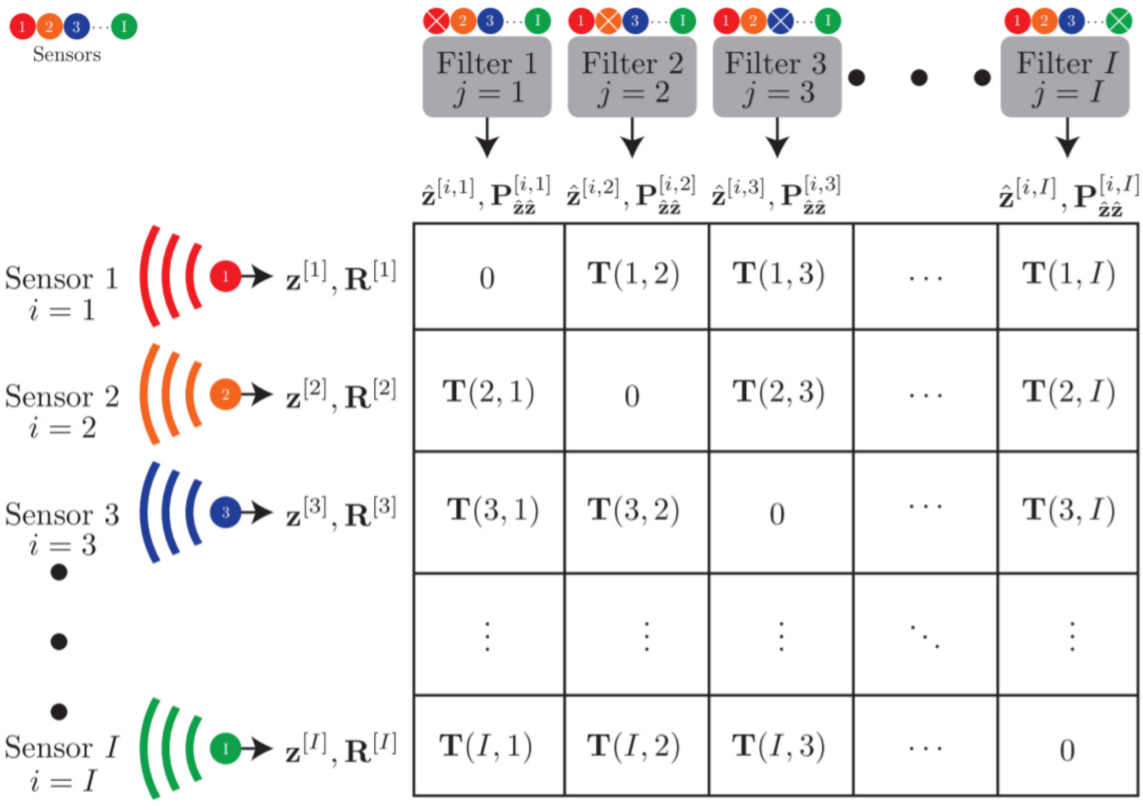

Figure 3: SAARM Test Matrix for i = 1...I Sensors [1]

3.3 Related Work

Although the Kalman filter algorithm can be traced back to 1960 [22], it is only until recently that the acceleration of the algorithm on computer systems has been explored. In 2011, Min-Yu Huang *et al.* [29] showed how the algorithm could be accelerated by parallelizing the Kalman filter matrix computations on a GPU. The study decomposed the algorithm into the individual math operations required for a full iteration and analyzed the clock cycles required for each operation. The matrix inversion operation was shown to be the most time-consuming, followed by matrix multiplication. Transferring these types of calculations to the GPU decreased computation times as the data dimensions increased from 1000 to 7000. Final results demonstrated a GPU implementation provided significant improvement over a CPU, achieving a maximum speedup of 7398x when the observation dimension was 6000 and the state dimension was 4500. This approach has been utilized on multiple different applications [30][31][32][33][34][35].

An alternate method was present by Rosen in [36] and [37], which breaks data dependencies by reordering the Kalman filter equations. This allows for an almost completely parallel algorithm implementation, however it is only applicable to specialized Kalman filter applications. Thein and Kaung showed a similar approach [38] could be used for general Kalman filter applications by computing decoupled equations on separate processors decreasing overall execution time of the algorithm.

Karimipour and Dinavahi presented a systematic process in [39] that also considered the application employing the Kalman filter algorithm to maximize the extraction of task and data parallelism. These portions of the software were then executed on a multi-core processor to accelerate the system. Results showed a total speed up of 15x over a sequential application.

Lastly, [40] and [41] are the studies most comparable to this research. Both studies

implemented multiple Kalman filter instances on a GPU. Results in [40] showed a 20 to 30 percent performance increase over a CPU. However, the study conducted by Yussiff *et al.* [40] disclosed few implementation details which in turn provided minimal contribution to this research work. On the other hand, the study done by Amamra and Aouf [41] detailed the GPU kernel code that was utilized, the CPU to GPU data transfer method, and the data layout in memory. This information bolstered the software design decisions in discussed in the ensuing sections.

This research is unique in that it builds off the development of the ARMAS framework [1], which uses Kalman filters to provide a navigation solution resilient to sensor failures. For this research work, a sensor can be any entity that provides navigation readings including a GPS satellite. Detecting sensor failures requires an array of multiple Kalman filter instances running in the background in addition to a main Kalman filter which is responsible for providing the navigation solution to the user. When a sensor failure occurs, the main Kalman filter is replaced by the appropriate Kalman filter in the array to exclude the failed sensor from the navigation solution. This requires additional logic and control routines that have not been implemented in Kalman filter optimization studies discussed in this section. This paper details the first C++ implementation of ARMAS and the earliest parallel processing of implementation of ARMAS on a GPU device. Finally, the research results will identify the sequential to parallel processing efficiently cross-over points in terms of system sensors, faults, and Kalman filters instances.

3.4 CPU Implementation

The sequential ARMAS implementation was programmed in C++. C++ is a complied language that supports low-level operations [3], allowing direct hardware interaction with zero-overhead abstraction [5][8]. Comparative studies have shown

this programming language feature enables C++ to achieve higher speed performance than other languages [10]. Additionally, C++ allows for the most memory-efficient program models with the least variability [11]. The performance benefits of C++ facilitates identifying a distinct boundary when it becomes more advantageous to perform computations on a GPU over a CPU.

The Eigen linear algebra library [43] was used for the matrix data containers. The high-level C++ interface provided easy-to-use data structures that have built-in input/output interfaces for simpler code readability. Although, the Eigen library is capable of performing the required computations needed for ARMAS on a CPU, its expression templates metaprogramming technique is not able to run directly on a GPU. For this reason, Intel Math Kernel Library [44] was used as the Basic Linear Algebra Subprograms (BLAS) library for the ARMAS CPU version. Using a BLAS library on both the CPU and GPU allows for a sequential vs parallel comparison of performance given that equivalent math routines are executed.

Pre-generated simulated sensor measurements were processed one at a time. The overall execution of the CPU program was kept purely sequential, avoiding the use of multi-threading and strictly using for-loops to iterate through computations. This allowed us to distinguish ARMAS code segments that are best suited for either sequential or parallel processing.

3.5 GPU Implementation

The parallel ARMAS design was constructed to run on an NVIDIA GPU using the CUDA computing platform model [45]. Early designs attempted the use of the OpenCL framework [46] as the programming interface for cross-platform modularity. Unfortunately NVIDIA limits OpenCL support to version 1.2 on their GPU devices reducing the range of capabilities to a sub-set of the latest OpenCL specification. To

take full advantage of the processing power on the available NVIDIA hardware, the GPU ARMAS program utilizes the CUDA application programming interface.

CUDA conveniently works with the C and C++ programming languages. This permitted the GPU code design to follow the same program logic flow as the CPU code with the added feature of being able to execute specified sections of code concurrently. The ARMAS GPU program was built to be as stand-alone as possible with minimal interaction with the CPU avoiding costly memory transfers. Additionally, this tested the concept of being able to run ARMAS unobstructed in the background freeing up main system resources in real-world application.

3.5.1 Parallelization Approach

3.5.1.1 Concurrent Filter Execution

The number of Kalman filters required to be maintained by an ARMAS system escalates quickly as system sensors and detectable simultaneous faults are scaled up. For example using Equation (55), a system with 40 active sensors that can detect 3 simultaneous faults requires 10,700 filters (40 filters for 1 fault, 780 filters for 2 simultaneous faults, and 9880 filter for 3 simultaneous faults) in addition to the main filter which outputs the navigation solution to the user. The sequential execution of computations required to maintain each filter lowers the possibility of running this size of ARMAS system on common cost-efficient computer systems, such as the CPU used in this study, for real-time use.

Performing these computations in parallel for each filter on a GPU device can accelerate the overall execution as illustrated in Figure 4, if the number of Kalman filters is large enough to hide the long latency memory accesses. To achieve this potential improvement, the propagate and update functions were converted into CUDA kernels, allowing each thread that is launch to correspond to a single Kalman filter.

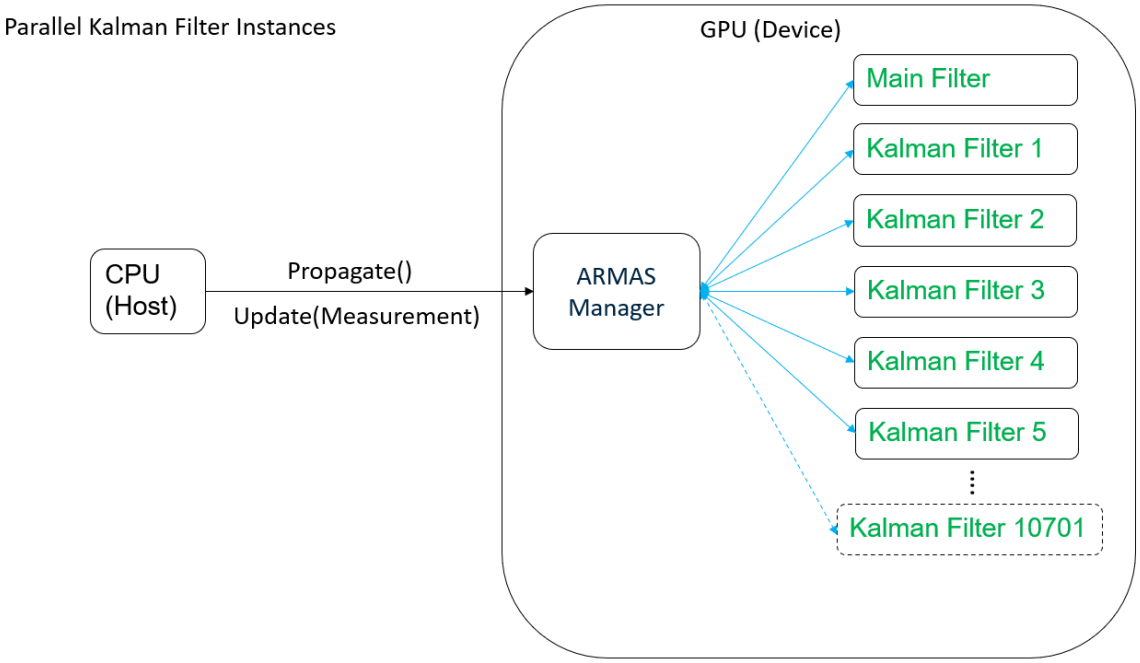

Figure 4: Parallel Kalman Filter Instances on a GPU device.

3.5.1.2 Concurrent Test Matrix Construction

A unique feature to ARMAS, when compared to other Kalman filter estimation software, is its ability to detect sensor failures during the update phase of the algorithm. ARMAS evaluates residual vector measurements for a specified time window to determine if sensor history measurement readings fail to be within an expected distribution. A test matrix is comprised of these results to attain a consensus which identifies the faulty sensor. A result is needed for all sensors in each filter as shown in Figure 5, where the blue arrows are the same set of computations performed on the different sensor data sets.

This collection of identical computations on different data elements becomes an ideal computing problem appropriate for GPU processing. The construction of this test matrix occurs as long as there is enough residual vector measurement data to fill the specified time window. Short specified time windows results in more test matrix generations in a given ARMAS system.

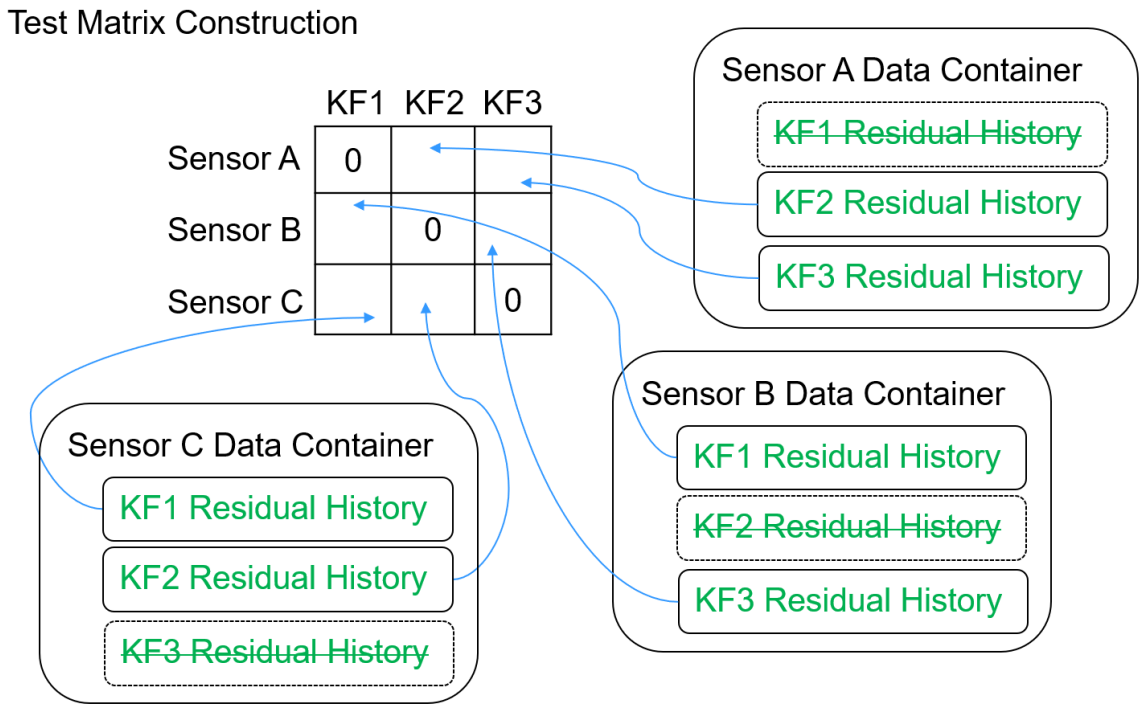

Figure 5: Test Matrix Construction Diagram

3.5.1.3 Parallel BLAS Execution

A finer-grain level of parallelism was obtained on the math operations performed within ARMAS with the help of the CUBLAS library [47]. CUBLAS is a parallel equivalent of the BLAS library used to perform matrix arithmetic on NVIDIA GPU devices [31]. This is achieved by partitioning data to different CUDA threads and computing the math routines concurrently, when possible. Such is the case with matrix operations like matrix-matrix addition and matrix-matrix multiplication.

3.6 Test Procedure

3.6.1 Testing Hardware

The test cases were evaluated on a Dell Precision 7720 workstation configured with a Intel Core I7-7920HQ CPU and an NIVIDA Quadro P5000 GPU. The Intel

Core I7-7920HQ operates at 3.10 GHz with 8 MB of Cache. The workstation system total RAM contained 32 GB. The NIVIDA Quadro P5000 GPU is comprised of 2048 CUDA Cores operating at 1.51 GHz with 16 GB of GDDR5 memory. The results presented in this paper are particular to this testing hardware. However, the parallel processing techniques discussed in this paper are valid optimization approaches for CUDA compatible hardware devices.

3.6.2 Test Setup

For all test scenarios, a $\dot{\mathbf{x}}(\mathbf{t})$ is a ten system state vector defined as

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} \dot{x_{p_x}}(t) & \dot{x_{p_y}}(t) & \dot{x_{p_z}}(t) & \dot{x_{v_x}}(t) & \dot{x_{v_y}}(t) & \dot{x_{v_z}}(t) & \dot{x_{a_x}}(t) & \dot{x_{a_y}}(t) & \dot{x_{a_z}}(t) & \dot{x_c}(t) \end{bmatrix}^T \quad (56)$$

where x_p is the vehicle's three-dimensional position in [m], x_v is the three-dimensional velocity in [m/s], x_a is the three-dimensional acceleration in [m/s^2], and x_c is a clock error bias.

The initial position state estimates were set to zero for the x and y direction and 200 [m] in the z direction. The initial velocities were 4.25 [m/s] for the x direction, 5.03 [m/s] for the y direction and zero [m/s] for the z direction. These values were randomly generated for the first test scenario and used for the subsequent test runs. The initial acceleration state estimates were set to zero. The initial clock error bias was also randomly generated for the first test scenario and again, used for the subsequent test runs. The initial randomly generated clock error bias was (4.4083×10^3) [m].

The initial state estimation error covariance was set to 10^2 [m^2] in position, 10^2 [m^2/s^2] in velocity, $(1\times10^{-2})^2$ [m^2/s^4] in acceleration, and 8000^2 [m^2] for the clock error bias. Each trial was propagated using $\Delta t_k = 0.5$ [s], starting at $\Delta t_k = 0$ [s] with induced faults occurring at $\Delta t_k = 300$ [s], and ending at $\Delta t_k = 600$ [s]. The

false alarm test significance level was set to $\alpha = 1/500000 = 2.0 \times 10^{-6}$, while the test epoch was set to M = 2400 samples, which is equivalent to 1200 [s], for all test experiments.

Each ARMAS subsystem used sensor models inherited from the original ARMAS implementation [1] that included one three-dimensional position sensor modeled by,

$$\mathbf{z_k^{[1]}} \;=\; \mathbf{s} \odot \mathbf{x_{p_k}} \;+\; \mathbf{v_k^{[1]}}, \tag{57}$$

$$\mathbf{v_k^{[1]}} \;\sim\; \mathcal{N}\left(\underset{\mathbf{3x1}}{\mathbf{0}}, \mathbf{100^2}\,\underset{\mathbf{3x3}}{\mathbf{I}}\right), \tag{58}$$

where $s = [s_x \; s_y \; s_z]^T$ is a three-dimensional scale factor, $\odot$ symbolizes the Hadamard product, and I is an identity matrix. Pseudorange sensors modeled by,

$$\mathbf{z_k^{[2...9]}} \;=\; |\mathbf{0t_{[2...9]}} - \mathbf{x_{p_k}}|\mathbf{0} + \;\mathbf{b_k} + \;\mathbf{v_k^{[2...9]}}, \tag{59}$$

$$\mathbf{v_k^{[2...9]}} \;\sim\; \mathcal{N}\left(\mathbf{0}, \mathbf{10^2}\right), \tag{60}$$

where t is a three-dimensional position of the satellite, x_{p_k} is the three-dimensional position of the vehicle at t_k and b_k is a First Order Gauss-Markov process simulating a simple receiver clock error with a time constant $\tau = 3600$ $[s]$ and $\sigma^2 = 8000^2$ $[m^2]$.

Lastly, a three-dimensional velocity sensor modeled by,

$$\mathbf{z_k^{[10]}} \;=\; \mathbf{s} \odot \mathbf{x_{v_k}} \;+\; \mathbf{v_k^{[10]}}, \tag{61}$$

$$\mathbf{v_k^{[10]}} \;\sim\; \mathcal{N}\left(\underset{\mathbf{3x1}}{\mathbf{0}}, \mathbf{50^2}\,\underset{\mathbf{3x3}}{\mathbf{I}}\right). \tag{62}$$

.

All ARMAS system test configurations included one three-dimensional position sensor, one three-dimensional velocity sensor, and pseudorange sensors that varied

in size as they scaled up in order to increase the computational load. For instance, the smallest system sensor size consisted of one three-dimensional position sensor, one three-dimensional velocity sensor, and eight pseudorange sensors that simulated GNSS satellites. To increase the computational load, the next system sensor configuration consisted of one three-dimensional position sensor, one three-dimensional velocity sensor, and 12 pseudorange sensors. This trend was continued until the largest system sensor size of 40 (one three-dimensional position sensor, one three-dimensional velocity sensor, 38 pseudorange sensors) was reached.

Each system sensor size configuration was constructed 3 times to respectively detect 1, 2, and 3 simultaneous fault(s), each requiring an increasing amount of Kalman filter that need to be maintained as shown by the table in Figure 6. This purses an ideal situation, where ARMAS identifies the most simultaneous faults as possible in a time step resulting in the most accurate state estimate result.

All test cases were initiated with warm up cycles to optimize and stabilize memory operations. After the warm up, 20 iterations of each test case were recorded for execution time and the results were averaged for a final value.

Total number of Kalmn Filters required			
Faults	1	2	3
NumSensors			
10	11	56	176
11	12	67	232
12	13	79	299
13	14	92	378
14	15	106	470
15	16	121	576
16	17	137	697
17	18	154	834
18	19	172	988
19	20	191	1160
20	21	211	1351
21	22	232	1562
22	23	254	1794
23	24	277	2048
24	25	301	2325
25	26	326	2626
26	27	352	2952
27	28	379	3304
28	29	407	3683
29	30	436	4090
30	31	466	4526
31	32	497	4992
32	33	529	5489
33	34	562	6018
34	35	596	6580
35	36	631	7176
36	37	667	7807
37	38	704	8474
38	39	742	9178
39	40	781	9920
40	41	821	10701

Figure 6: Kalman filters required by sensor size and detectable simultaneous faults. Blue indicates the test cases that were performed.

3.7 Results

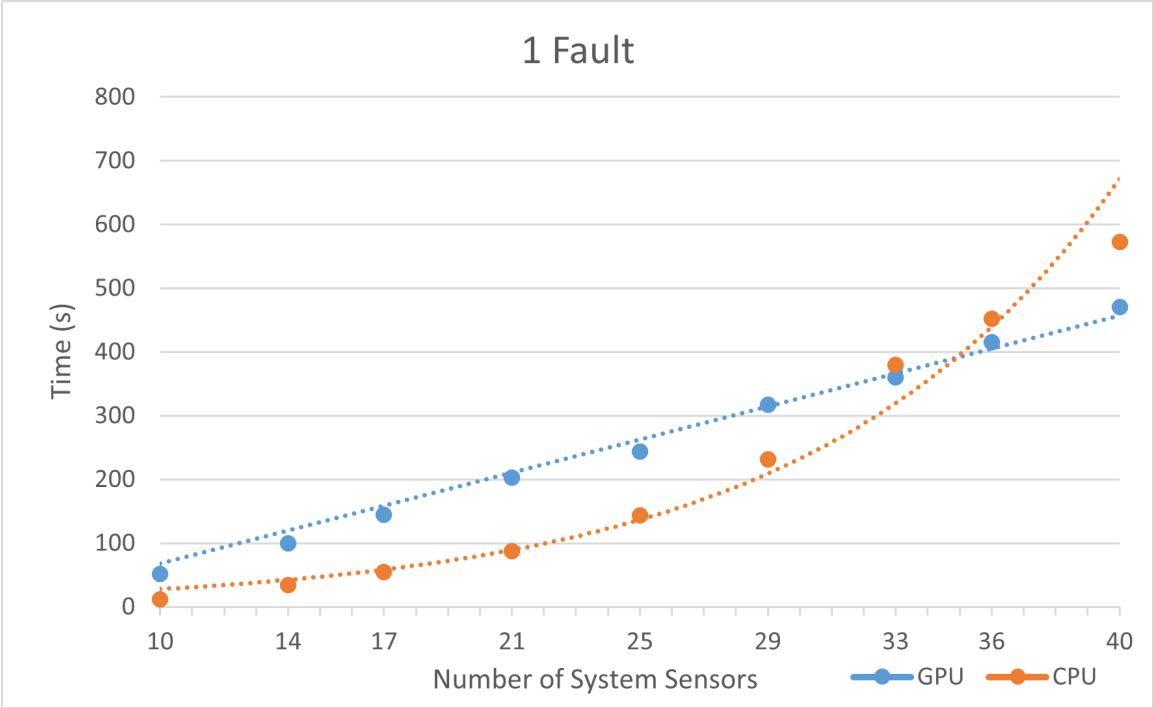

Figure 7: Total program run time test results for an ARMAS system configured to detect at most one sensor fault.

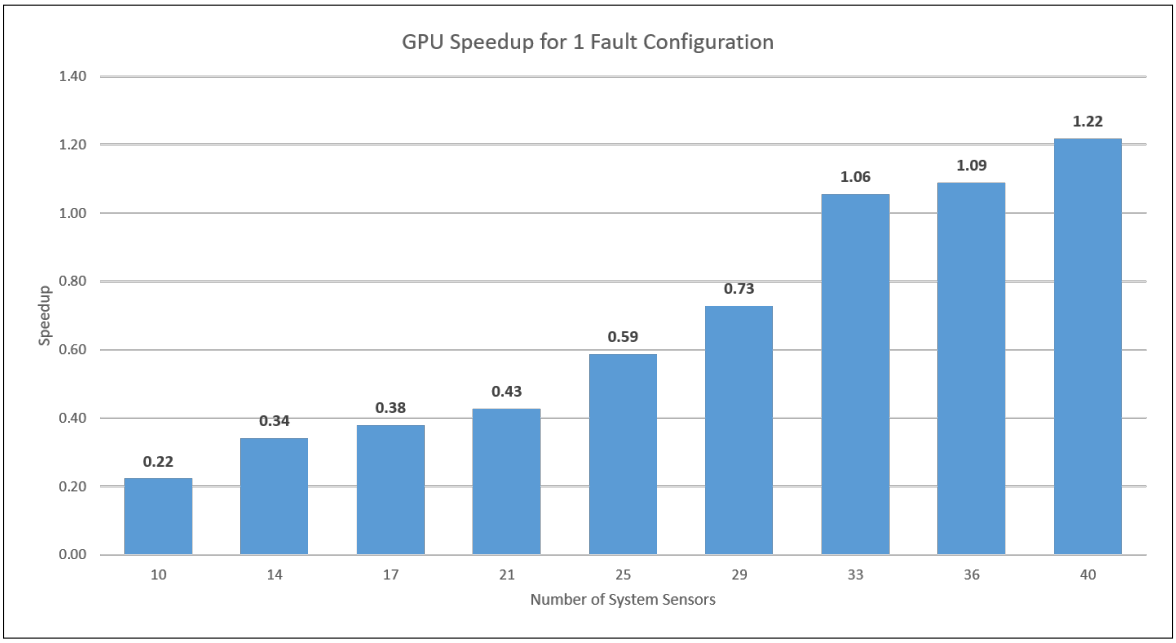

Figure 8: Speedup achieved for 1 fault ARMAS configuration.

Test results showed smaller system sensor configurations to be faster on a CPU. As the system sensors size and number of faults increased, the GPU exhibited greater efficiency with the increased computational load. Figure 7 shows this efficiency cross-over point from the CPU to GPU for a full system simulation occurs when the sensor size is 33 and detectable faults is one. Figure 9 shows the efficiency cross-over point occuring when the sensor size is 25 and the detectable faults is two. Figure 11 shows this point occurs when the sensor size is 21 and detectable faults are three.

Figures 8, 10, and 12 show the achieved speed up with each full test case. The GPU ARMAS implementation was able to achieve a approximate maximum speedup of 3.5x in total execution time over the CPU in the most stressing case of a 40 sensor system configured to detect at most three simultaneous faults as shown in Figure 12. Although the speedup is significant, it can be assessed that the ARMAS version created for this study is unsuitable for real-time applications after the sensor size is greater than 21 sensors for system designed to detect three or more simultaneous faults on the current hardware. This information is represented in Figure 13, where the green color represents faster ARMAS CPU times, orange represents faster ARMAS GPU times, and red shows execution times over real-time application constraints. Similarly, we can estimate this software will reach this same overexertion point for real-time constraints when the system sensor size is 93 sensor for the one fault case and 45 sensors for the two simultaneous faults case. Section 3.8 addresses potential methods to further accelerate this software.

Notably, software timer measurements revealed the construction of sensor fault test matrix was able to exploit advantages from parallelizing the computing tasks as the test matrix elements scaled up to a size of 1600, ultimately achieving a 1.44 speedup over the CPU implementation as shown in Figure 15. The GPU test matrix building function was able to achieve a logarithmic growth when a system was con-

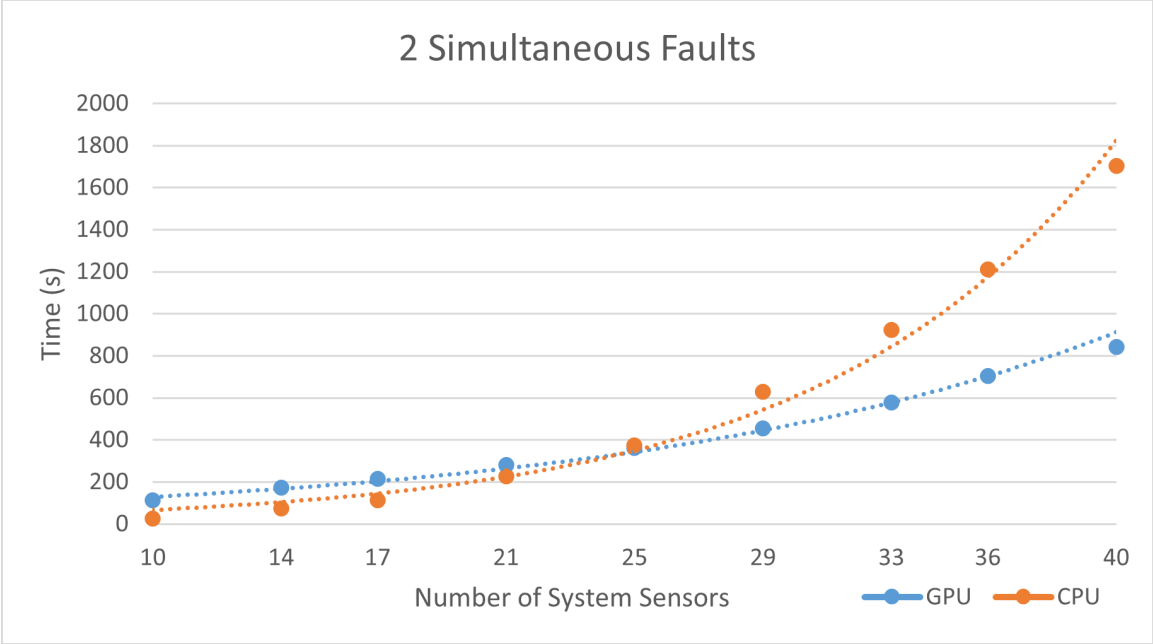

Figure 9: Total program run time test results for an ARMAS system configured to detect at most two simultaneous sensor faults.

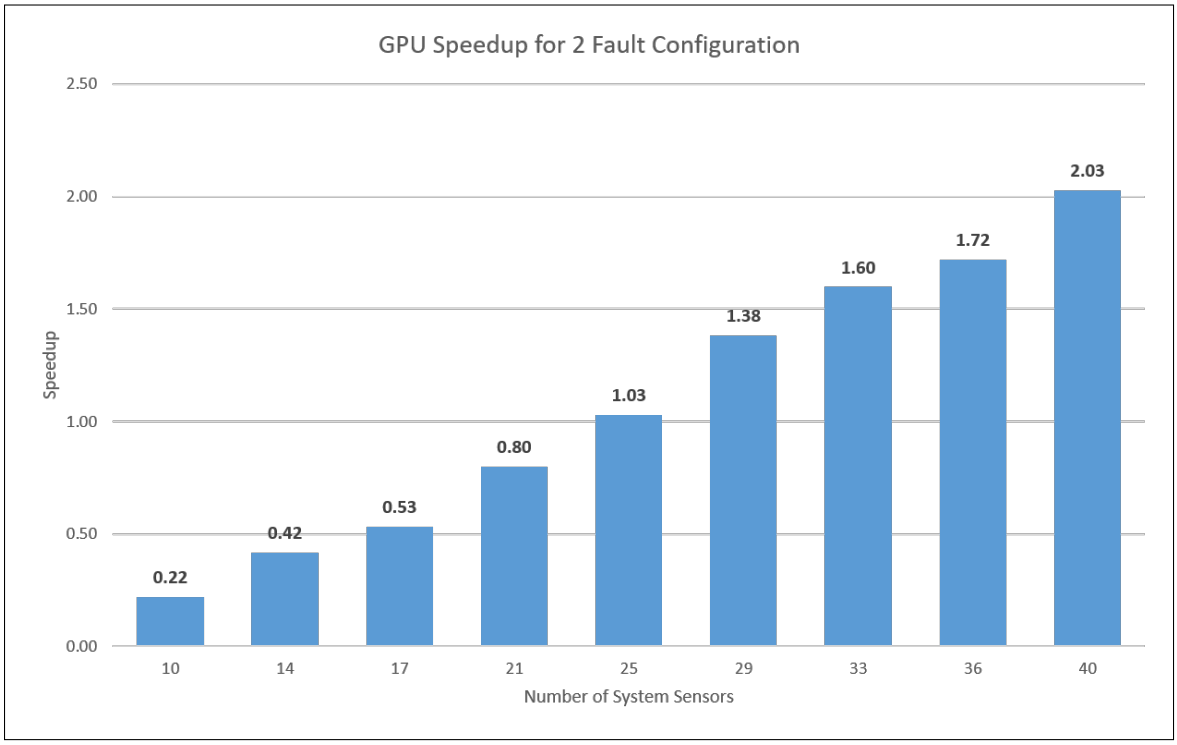

Figure 10: Speedup achieved for 2 fault ARMAS configuration.

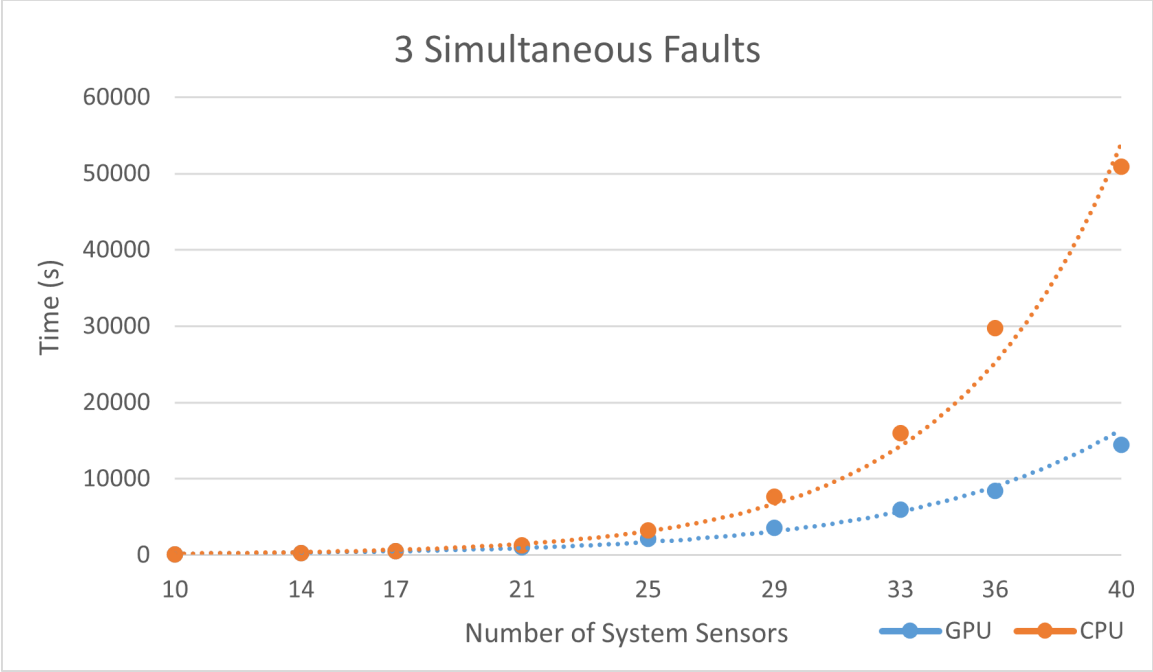

Figure 11: Total program run time test results for an ARMAS system configured to detect at most three simultaneous sensor faults.

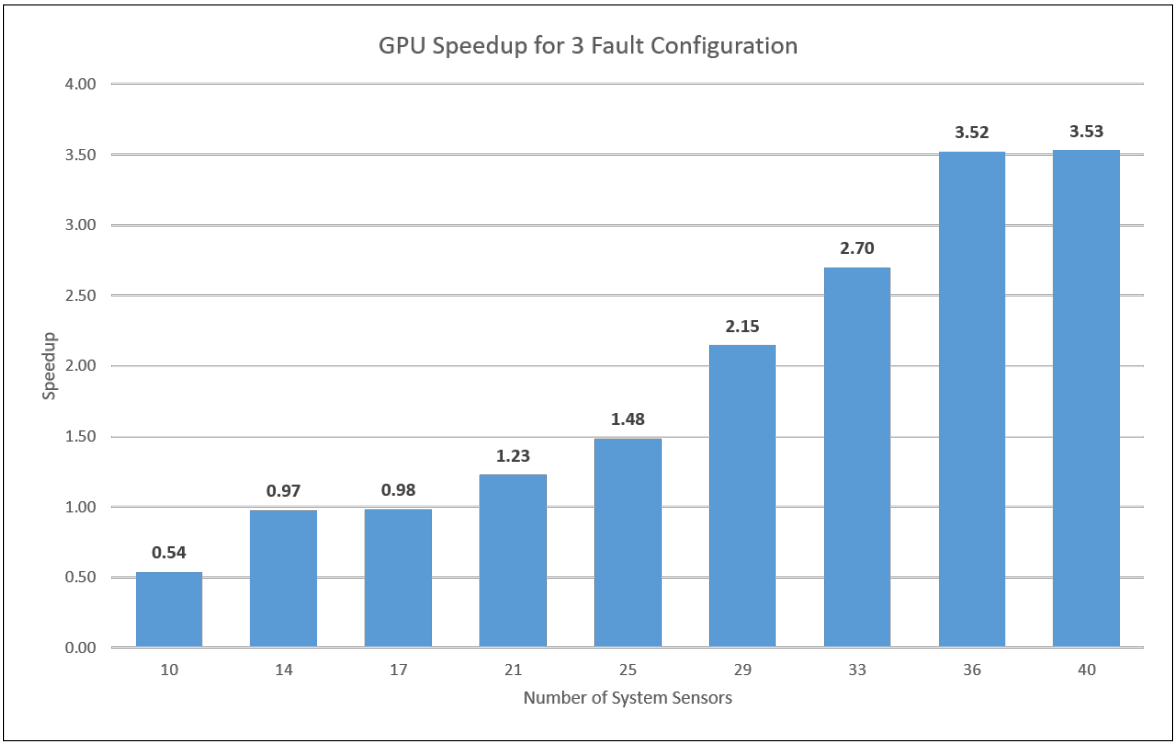

Figure 12: Speedup achieved for 3J fault ARMAS configuration.

Total number of Kalmn Filters required			
Faults	1	2	3
NumSensors			
10	11	56	176
11	12	67	232
12	13	79	299
13	14	92	378
14	15	106	470
15	16	121	576
16	17	137	697
17	18	154	834
18	19	172	988
19	20	191	1160
20	21	211	1351
21	22	232	1562
22	23	254	1794
23	24	277	2048
24	25	301	2325
25	26	326	2626
26	27	352	2952
27	28	379	3304
28	29	407	3683
29	30	436	4090
30	31	466	4526
31	32	497	4992
32	33	529	5489
33	34	562	6018
34	35	596	6580
35	36	631	7176
36	37	667	7807
37	38	704	8474
38	39	742	9178
39	40	781	9920
40	41	821	10701

Figure 13: ARMAS execution time color diagram. Green represents faster CPU times, orange represents faster GPU times, and red are execution times over real-time application constraints. Blue indicates the test cases that were performed.

figured to detect one fault as shown in Figure 14. Alternatively, the CPU test matrix building function grew linearly for the same one fault case. The CPU remained faster up until the matrix element size reached 1102. Matrices larger 1102 achieved quicker matrix construction times on the GPU. This optimization is primarily responsible for faster GPU execution times seen in the Figures 7 and 9, where the total number of Kalman filter in the ARMAS system remains below 1000.

Furthermore, parallelizing the Kalman filter instances saw benefits after the filter array size grew larger than 1005 as shown in Figure 16. The time to complete one sensor measurement update for all system filters on the CPU grew at an approximate exponential rate as the filter array size increased, while remaining practically linear on the GPU for systems containing up to 10701 filters. This improvement becomes the main execution time enhancement in the three simultaneous fault case as calculating Equations (42) through (44) for all system filters becomes the most time consuming task of the program. This can be seen by the 4X speedup achieved in the ARMAS instance that maintained 10701 Kalman filters as shown in Figure 17.

Finally, the ARMAS system state logic demonstrated no improvement by off-loading this work on the GPU. The CPU ARMAS system state logic run time was between 2-3x faster on average than the GPU ARMAS logic run time, but overall this is a negligible difference as the CPU system state logic accounted for less than one percent of the overall execution time and GPU state logic was less then two percent.

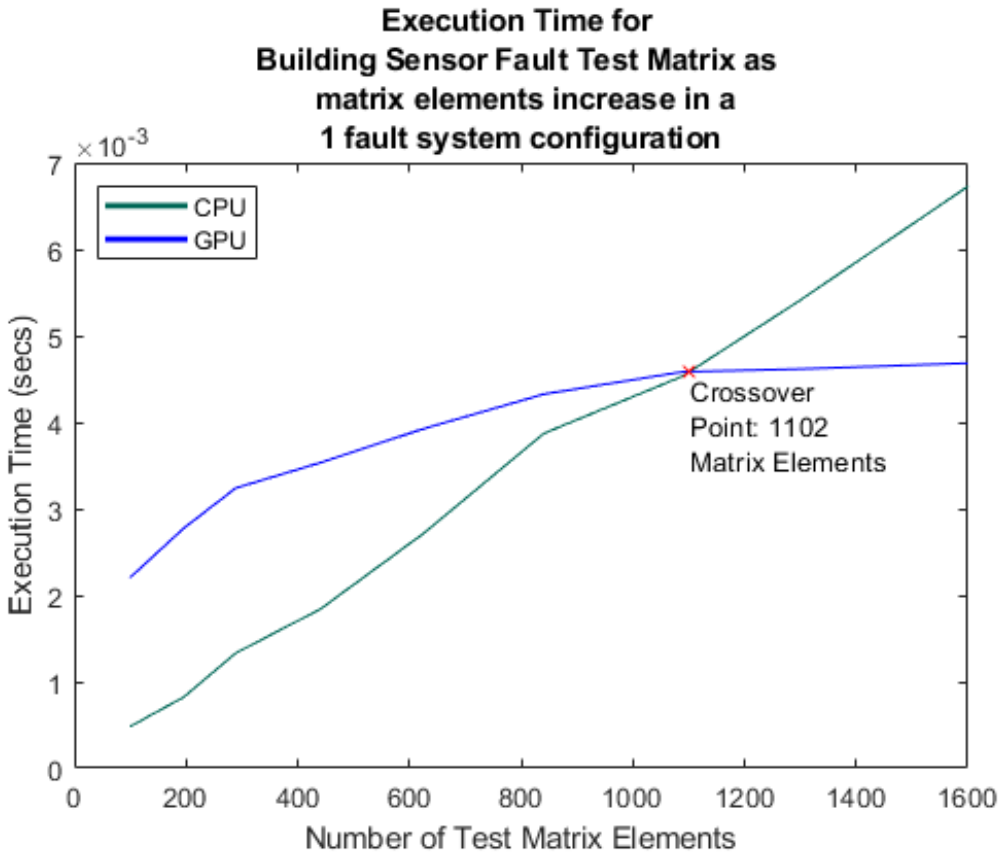

Figure 14: ARMAS execution time graph for building the sensor fault test matrix.

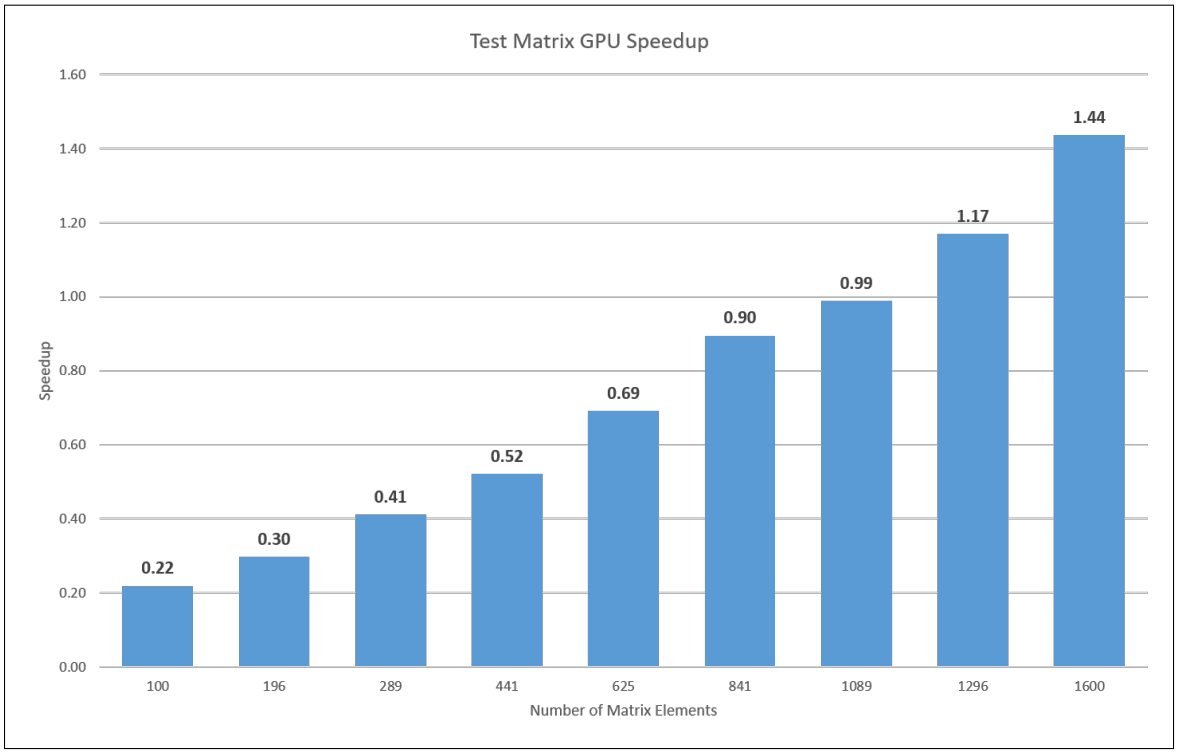

Figure 15: Speedup for building the sensor fault test matrix.

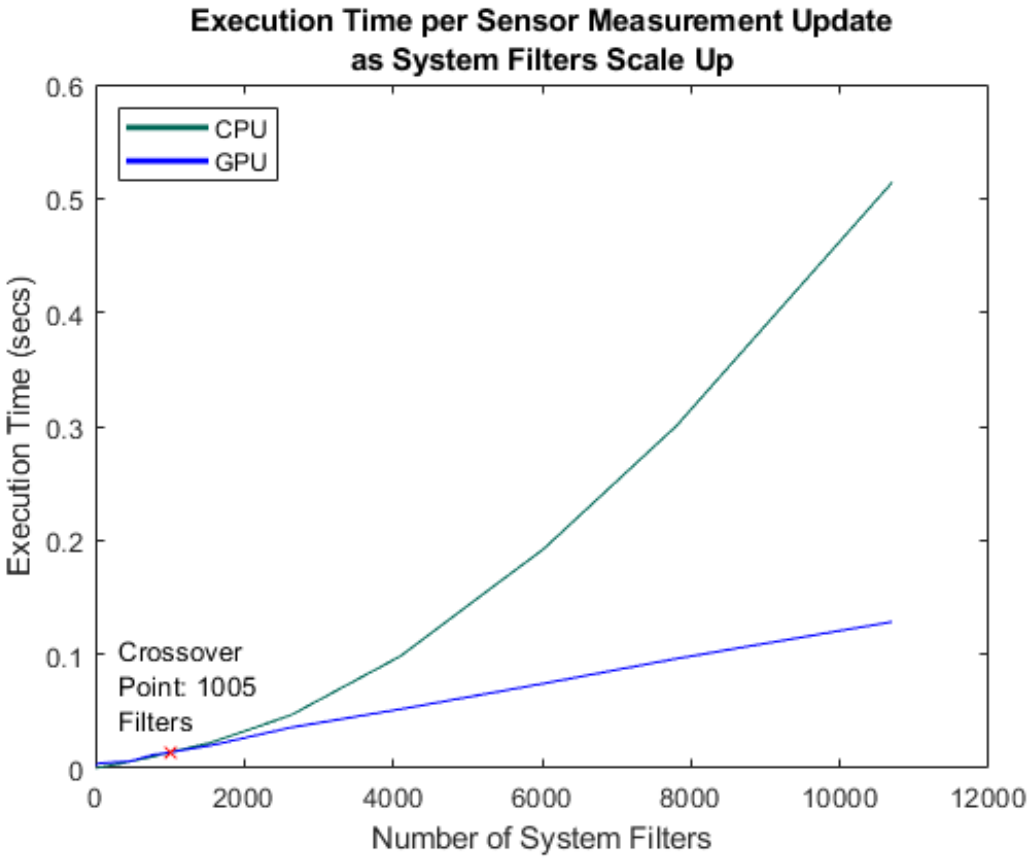

Figure 16: ARMAS execution time per sensor measurement graph.

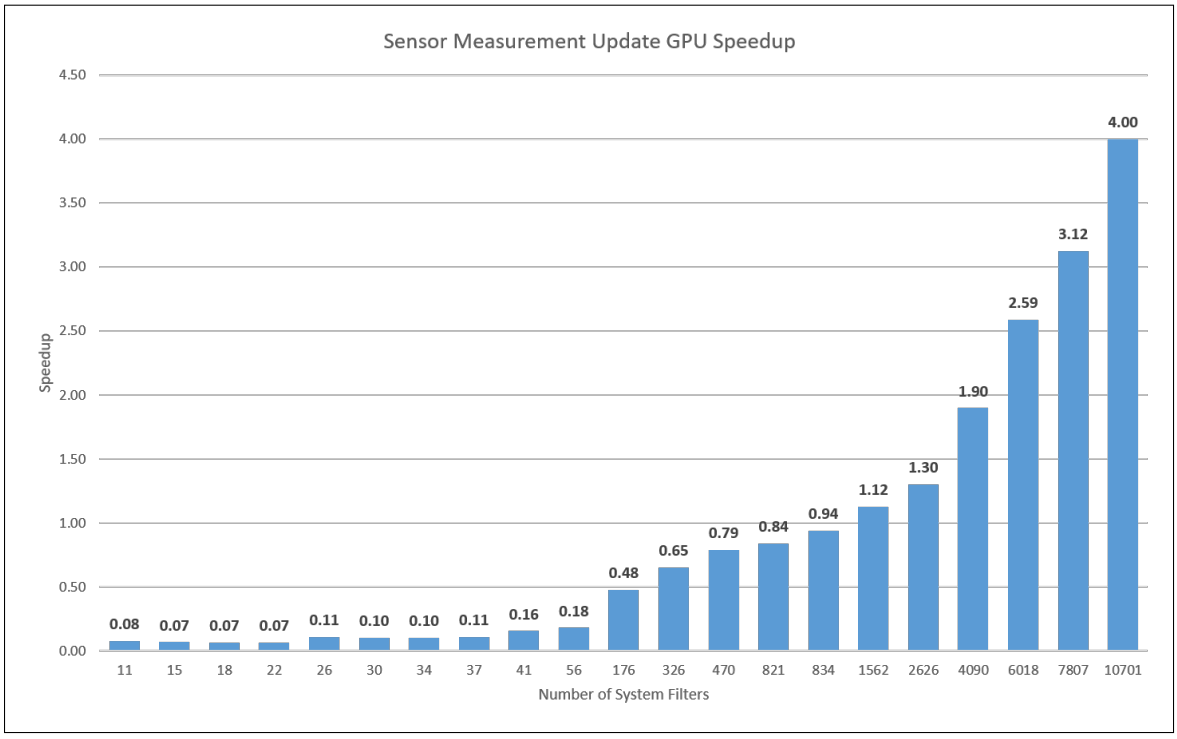

Figure 17: Speedup per sensor measurement graph.

3.8 Future Works

Throughout this study, the following key areas were identified as potential follow-on work.

3.8.1 Improved GPU eigen solver

Developing an ARMAS GPU implementation that was largely a stand-alone GPU program proved much harder than expected given the set of GPGPU development tools available. The goal was to be able to launch a CUDA thread and have the program run autonomously with minimal interaction with the CPU. This requires most of the code to be called strictly from the device side. Conversely, the traditional GPGPU Heterogeneous computing model relies on the CPU to initiate function calls from the host side. This limited the available libraries that could be used out of the box. The device side cuBLAS library supported in CUDA 9.0 was used for most of the required computations, but the cuSOLVER library needed to solve for the eigenvalues and eigenvectors of matrix is only a host callable library. Early attempts to adapt the existing cuSOLVER code base for our purposes lead to constant compiler errors that ultimately became unsolvable with the available time. As an alternative, an eigen solver using the Jacobi eigenvalue algorithm obtained from Dr. John Burkardt web-page [48] was modified to run as CUDA function. This provides an area of improvement in future ARMAS GPU implementations by simply altering the eigen solver code to the more efficient QR algorithm.

3.8.2 Implement Traditional Heterogeneous Computing Model

As stated in the previous subsection, developing a stand-alone GPU program is difficult as it is usually not optimal for most cases. Complex control logic operations perform better on CPUs, while GPUs should be mainly reserved for tasks with high

computational cost and high data parallelism. Extracting the logic control currently implemented on the device side for the purposes of this study and off-loading it to the CPU while keeping the BLAS computations on the GPU, allows the specified hardware to run the software more efficiently at the cost of using up more CPU resources and requiring expensive memory transfers.

3.8.3 Smarter Configuration of Kalman Filters

Taking into account an operational wartime scenario, one can assume a batch of sensors in an ARMAS system could be a satellite constellation. Additionally, the assumption can be made that a constellation of satellites normally all operate under the same frequency bands. Therefore, under a jamming situation, one can expect that entire constellations would be unavailable to user as opposed to arbitrary satellites. Using this rationale, we can avoid the burden of detecting continuous simultaneous faults by immediately disregard satellites in the same constellation that begin to show measurement errors. This type of logic can be achieved by having multiple ARMAS subsystems in one application. Each ARMAS subsystem would represent a different GNSS constellation that would be configured to detect a manageable number of faults (i.e. one fault detection). If more than the specified faults occur, then the integrity of the ARMAS subsystem fails excluding that set sensors from the navigation solution. Initial proof of concept trails have shown lower computational loads while maintaining an acceptable level of accuracy.

3.9 Conclusion

In conclusion, this research presented an approach that parallelized Kalman filter instances and other computation tasks to accelerate the ARMAS framework. Test cases incrementally scaled up the system sensor size and detectable system fault to

evaluate computational load and measure execution time. This study provided a deep understanding of the ARMAS processing power requirements. Small sensor size configurations proved to be faster on a CPU, while an approximate 3.5 order of magnitude speedup was achieved by the GPU ARMAS implementation over the CPU version on a large sensor system configuration. The information lead to the system modeling of computing loads with respect to sensor size and detectable sensor faults. Results show promise that a real-time ARMAS system can be achieved for large scale applications through parallel processing on many-core architectures.

IV. Scholarly Article: Fusing multiple ARMAS frameworks for an overall lower computational demand

Abstract

Alternative navigation is an area of research that employs a variety of sensor technologies to provide a navigation solution in Global Navigation Satellite System degraded or denied environments. The Autonomy and Navigation Technology Center at the Air Force Institute of Technology has recently developed the Autonomous and Resilient Management of All-source Sensors (ARMAS) navigation framework which employs an array of Kalman filters to provide a navigation solution resilient to sensor failures. The Kalman filter array size increases exponentially as system sensors and detectable faults are scaled up, which in turn increases the computational power required to run ARMAS in a real-world application. In an effort to engineer a real-time ARMAS system, this study presents a novel approach that utilizes multiple ARMAS subsystems to generate a navigation solution with less computational demand.

4.1 Introduction

Alternative navigation is an area of research that employs a variety of sensor technologies to provide a navigation solution in Global Navigation Satellite System (GNSS) degraded or denied environments [42]. The Autonomy and Navigation Technology (ANT) Center at the Air Force Institute of Technology (AFIT) has recently developed the Autonomous and Resilient Management of All-source Sensors (ARMAS) navigation framework, which produces a navigation solution resilient to sensor failures [1]. ARMAS employs a sensor fault detection and exclusion procedure that utilizes a bank of overlapping, but unique, Kalman filters which identify faulty sensors through a general consensus. One key benefit of this procedure is that an uncorrupted

Kalman filter always exists within an ARMAS system properly configured to handle sensor fault case.

To transition the academic software closer to an operational real-time system, the nontrivial computational power required to run ARMAS must be addressed, as demonstrated in Chapter III. Specifically, the initial ARMAS design utilizes an array of unique Kalman filters which identify faulty sensors. The size of the Kalman filter array depends on the number of system sensors and the desired detectable simultaneous sensor faults specified by the user. As a general approximation, the Kalman filter array size increases exponentially as system sensors and detectable faults are scaled up linearly, which in turn increases the computational load required to run ARMAS in a real-world application.

In an effort to engineer an operational real-time ARMAS system, this study presents a novel approach which utilizes multiple ARMAS subsystems to generate a navigation solution with less computational demand while still providing the same levels of navigation integrity. This research takes into account a hypothetical operational scenario where it can be assumed that the sensors in an ARMAS sub-system are composed of the psuedoranges from a single frequency of a single satellite constellation, e.g. GPS L1 pseudoranges grouped in one ARMAS sub-system. Under a jamming situation, one would then expect that the entire constellation operating in the targeted frequency band would be unavailable to the user, thus causing a simultaneous failure of many sensors.

Using this rationale, the burden of detecting continuous simultaneous faults can be avoided by immediately disregarding satellites in the same constellation and frequency band that begin to show measurement errors. This type of logic can be achieved by having multiple ARMAS subsystems in one application. Each ARMAS subsystem would represent a different GNSS constellation frequency band that would

be configured to detect a manageable number of sensor faults (i.e. one or two fault detection). If more than the specified faults occur, then the integrity of the ARMAS subsystem fails and the overall framework can exclude the grouped set of sensors from the navigation solution.

Conversely, the current ARMAS model is a single system architecture that must detect and identify every faulty sensor that is to be excluded from the navigation solution. This creates a high computational demand when multiple simultaneous sensor faults must be detected and identified. This paper details an ARMAS subsystem model implementation and the results of a simulated scenario.

The rest of this paper is organized as follows. Section 4.2 provides background information, section 4.3 summarizes related work, and section 4.4 provides system implementation details. Section 4.5 and 4.6 presents the test scenario and initial results. Lastly, section 4.7 and 4.8 discuss future works and the main conclusions of this work.

4.2 Background

4.2.1 Kalman filter

The Kalman filter, developed by Rudolf E. Kalman in 1960 [22], is used to perform recursive state estimation for a continuous-time process model in the form,

$$\dot{\mathbf{x}}(\mathbf{t}) \;=\; \mathbf{F}\mathbf{x}(\mathbf{t}) \;+\; \mathbf{B}\mathbf{u}(\mathbf{t}) \;+\; \mathbf{G}\mathbf{w}(\mathbf{t}), \tag{63}$$

where $\dot{\mathbf{x}}$ is the system state vector, $\mathbf{u}$ is the system input control vector, and $\mathbf{w}$ is the white noise components vector. $\mathbf{F}$, $\mathbf{B}$, and $\mathbf{G}$ are linear operator matrices with constant coefficients for the state vector, control input vector, and noise vector, respectively. The discretization of the Kalman filter equations using the Van Loan

method [23], allows the algorithm to be used in modern navigation systems to estimate a vehicle's system states.

Linear discrete sensor measurements modeled by

$$\mathbf{z_k} \;=\; \mathbf{H x_k} \;+\; \mathbf{v_k}, \tag{64}$$

where, $\mathbf{z}$ is the sensor measurement, $\mathbf{H}$ is the observation model which maps measurements to the states, and $\mathbf{v}$ is the white noise function, are used to update the state estimate at time point $\mathbf{k}$.

The Kalman filter algorithm is performed by iterating over 2 phases: propagate and update. The Kalman filter propagate phase the numerical integration of (63) for a state state estimate and state error covariance from timepoint $\mathbf{k}$ to time point $\mathbf{k}+1$, using

$$\hat{\mathbf{x}}^{-}_{\mathbf{k+1}} = \; \Phi \hat{\mathbf{x}}^{+}_{\mathbf{k}} \;+\; \mathbf{B_d u_k}, \tag{65}$$

$$\mathbf{P}^{-}_{\mathbf{k+1}} = \; \Phi \mathbf{P}^{+}_{\mathbf{k}} \Phi^{\mathbf{T}} \;+\; \mathbf{Q_d}, \tag{66}$$

where $\mathbf{x}$ is the state estimates, Φ is the discrete state transition, and $\mathbf{P}$ is the associated state error covariance matrix. The states and covariance are updated by comparing actual measurements with estimated measurements modeled by (64) and using the the Kalman gain, $\mathbf{K}$, using

$$\mathbf{K_k} = \; \mathbf{P^{-}_k H^T} \left[\mathbf{H P_k H^T + R} \right]^{-\mathbf{1}}, \tag{67}$$

$$\hat{\mathbf{x}}^{+}_{\mathbf{k}} = \hat{\mathbf{x}}^{-}_{\mathbf{k}} \;+ \mathbf{K_k} \left[\mathbf{z_k} - \; \mathbf{H} \hat{\mathbf{x}}^{-}_{\mathbf{k}} \right], \tag{68}$$

$$\mathbf{P}^{+}_{\mathbf{k}} = (\mathbf{1} \;-\; \mathbf{K_k H}) \mathbf{P}^{-}_{\mathbf{k}}, \tag{69}$$

where, $\mathbf{R}$ is the measurement error covariances.

4.2.2 Extended Kalman Filter

Systems that use non-linear dynamics or measurement models may not always achieve optimal results from implementing the linear Kalman filter equations. In certain cases, the non-linear systems can be transformed in linear approximations to produce accurate solutions. In these cases, the non-linear systems employ the Extended Kalman filter (EKF) [25][26] which uses the non-linear system dynamics equation in the form,

$$\dot{\mathbf{x}}(\mathbf{t}) = \mathbf{f}[\mathbf{x}(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}] + \mathbf{G}(\mathbf{t})\mathbf{w}(\mathbf{t}), \tag{70}$$

where $\mathbf{f}$ is a vector that contains functions that represent the system. The non-linear measurement model is given by,

$$\mathbf{z_k} = \mathbf{h}[\mathbf{x_k}, \mathbf{t_k}] + \mathbf{v_k}, \tag{71}$$

where $\mathbf{h}$ is a vector of functions that models the system sensor. To linearize the non-linear system, the states are converted using the perturbation model given by,

$$\delta\mathbf{x}(\mathbf{t}) \stackrel{\Delta}{=} \mathbf{x}(\mathbf{t}) - \hat{\mathbf{x}}(\mathbf{t}), \tag{72}$$

where $\delta\mathbf{x}(\mathbf{t})$ is the difference between the state estimate and the true state vector. To propagate the system to the time of the next measurement, the EKF integrates the non-linear dynamics equation over the time difference using

$$\hat{\mathbf{x}}_{\mathbf{k+1}}^{-} = \int_{\mathbf{t_k}}^{\mathbf{t_{k+1}}} \mathbf{f}[\mathbf{x}(\mathbf{t}), \mathbf{u}(\mathbf{t}), \mathbf{t}]\mathbf{dt} + \hat{\mathbf{x}}_{\mathbf{k}}^{+}. \tag{73}$$

The state covariance propagation does not change and continues using Equation (69), while the linearized dynamics model matrix is described by

$$\mathbf{F_k} = \left.\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right|_{\hat{x}_k^+}. \tag{74}$$

To update the state estimates using sensor measurement which may be non-linear, the measurement is initially predicted by evaluating the measurement model equation with the most recent estimate using

$$\hat{\mathbf{z}}_\mathbf{k} = \mathbf{h}[\hat{\mathbf{x}}_\mathbf{k}^-, \mathbf{t_k}], \tag{75}$$

$$\delta \mathbf{z_k} = \mathbf{z_k} - \hat{\mathbf{z}}_\mathbf{k}, \tag{76}$$

where $\delta \mathbf{z_k}$ is named the pre-update measurement residual which describes the difference between the actual and predicted measurements. By linearizing the non-linear measurement function $\mathbf{h}$ using

$$\mathbf{H_k} = \left.\frac{\partial \mathbf{h}}{\partial \mathbf{x}}\right|_{\hat{x}_k^-} \tag{77}$$

where H is the linearized matrix, the propagated and measured state estimates can be combined by calculating the Kalman gain using Equation (67), reducing the measurement update equation to

$$\delta \hat{\mathbf{x}}_\mathbf{k}^+ = \mathbf{K_k} \delta \mathbf{z_k}, \tag{78}$$

by utilizing perturbation state estimates and measurements.

The perturbation state $\delta \hat{\mathbf{x}}$, starts at zero during each filter iteration and is updated using Equation (78). It is then added to the nominal trajectory to produce a nominal estimate. The perturbation state is then reset to zero after each filter iteration.

4.2.3 ARMAS

ARMAS is a software framework that provides autonomous sensor management for navigation systems [1]. The ARMAS framework for this study is primarily designed around the Extended Kalman filter but can be adapted to other similar filtering techniques, such as the traditional Kalman filter or unscented Kalman filter. The state estimates are propagated by using the state dynamics process model and updated using measurements from $i = 1...I$ available sensors as described in (65)-(69).

ARMAS provides sensor failure resiliency through statistical residual monitoring based on a Mahalanobis distance likelihood function. This approach requires ARMAS to maintain an array of uniquely configured Kalman filters to detect sensor failures in addition to a main filter that produces the navigation solution for the user. To detect sensor failure(s) in a system, a set of J uniquely configured Kalman filters obtain a consensus output that identifies the faulty sensor(s). To detect a single sensor fault, $J = I$ (number of system sensors). In this situation, each Kalman filter contains a unique subset of $I - 1$ sensors, thus allowing a different sensor to be excluded in each filter. For this condition, under the assumption that at most one sensor can fail for this case, at least one of the J Kalman filters will be completely unaffected by the faulty measurement. The faulty sensor procedure populates a test results matrix, T, with

$$\mathbf{T(i,j)} = \begin{cases} \mathbf{0}, \text{Sensor i not associated with filter j} \\ \mathbf{0}, \text{No Fault Detected} \\ \mathbf{1}, \text{Fault Detected} \end{cases} \tag{79}$$

as illustrated in Figure 18.

For ARMAS to detect and manage multiple simultaneous faults, multiple banks of sub-filters must be maintained. The number of concurrent sub-filters, J_N, required

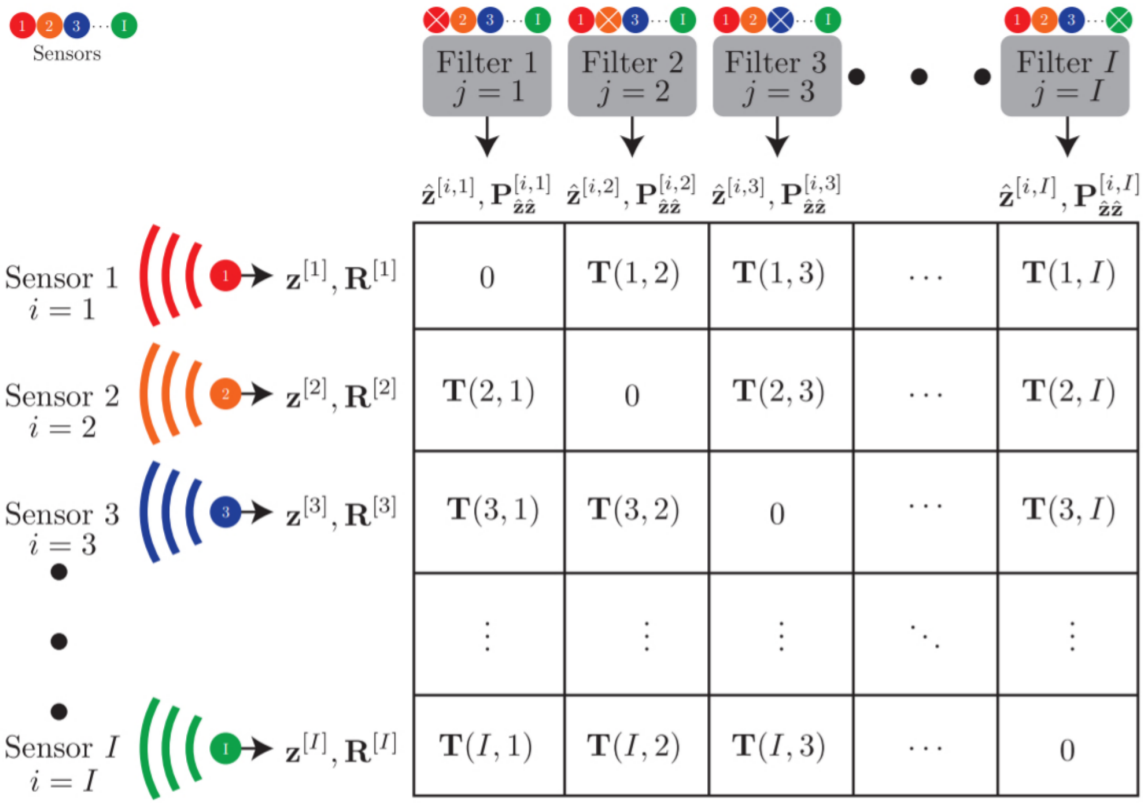

Figure 18: SAARM Test Matrix for i = 1...I Sensors [1]

to handle N simultaneous faults for I sensors is calculated using

$$\mathbf{J_N} = \binom{\mathbf{I}}{\mathbf{I-N}} = \frac{\mathbf{I!}}{\mathbf{N!(I-N)!}}. \tag{80}$$

So, for example, a three sensor ARMAS system set up to detect up to two simultaneous faults would require a array of Kalman filters configured as shown in Figure 19. Note that Equation (80) is used twice to calculate the number of filters needed for when $N = 1$ and $N = 2$, in addition to a main filter resulting in seven total system Kalman filters.

In this example, all three sensors are initialized as "trusted" and thus included in the main filter, which outputs the navigation solution to the user. Kalman filters 1 - 3, all have a unique sensor missing allowing the detection and identification of

a single sensor fault. In the scenario where two simultaneous sensor faults occur, Kalman filters 1 - 3 would all be affected and the ARMAS system logic would continue checking the next bank of Kalman filters to try and find a single unaffected filter. The unaffected filter identifies the culprits through the missing sensors in that particular Kalman filter. Figure 19 shows Kalman filters 4 - 6 with two unique sensors missing for the two simultaneous sensor faults scenario in this example.

When a sensor fault is detected and identified, the faulty sensor is removed from the main Kalman filter and thus also removed from the navigation solution, until the sensor recovers and verified as fully operational. If more than two simultaneous sensor faults occur in a system only configured to detect one sensor fault, the integrity of the ARMAS subsystem fails and the user is alerted. If more than one Kalman filter in the array is unaffected, ARMAS alerts the user that a sensor(s) fault(s) has occurred but is unable to identify the culprit(s) and continues regular operation until the sensor(s) can be identified.

Using Equation (80) as shown in section 4.2.3, one can calculate that the computing power required as sensors and faults are scaled up becomes significant very quickly. For example, using the test scenario later described in section 4.5 and Equation (80), a single ARMAS architecture with 26 sensors that is configured to detect up to 8 simultaneous faults would require 2,533,987 Kalman filters to be maintain throughout the lifetime of the application! This research seeks to address this compu-

	Kalman filter array						
Sensor	Main	KF1	KF2	KF3	KF4	KF5	KF6
1	•		•	•			•
2	•	•		•		•	
3	•	•	•		•		

Figure 19: Kalman filter array configuration for system detecting two simultaneous faults.

tation requirement by utilizing multiple ARMAS subsystems to generate a navigation solution with less Kalman filters to maintain.

4.3 Related Work

The detection and exclusion of sensor faults has been widely studied through different facets. Conventional methods used to achieve multi-sensor fault detection and exclusion often include the statistical interpretation of redundant snapshot measurements, solution separation vectors, or filtered residuals. The redundant snapshot measurements method presented in [49][50][51] use inconsistencies between calculated solutions based on measurement subsets from various GNSS satellites that form a linear least-squares solution matrix to detect and potentially identify the faulty sensor. The solution separation vectors approach seen in [52][53][54][55][56][57][58] evaluate the difference between horizontal position estimates and covariances for fault detection and uses a bank of parallel sub-filters for faulty sensor identification. Lastly, filtered residual methods outlined in [59][60][61][62] averages time sequence Kalman filter residuals to detect a sensor fault and identify the culprit.

Studies in [63][64][65] are the most comparable sensor fault detection methods to ARMAS, which specifically utilize a bank of Kalman filters to monitor test statistics. If the test statistics fall outside an expected chi-square cumulative distribution function threshold, a fault alarm is produced. Although these are similar works, the method used in ARMAS referred to as Sensor-Agnostic All-source Residual Monitoring (SAARM) is unique in that it:

- Does not constrain faults to only biases,
- Can be scaled for multiple simultaneous sensor faults,
- Detects faults (sensor model mismatches) in and across multiple domains,

- Does not require simultaneously redundant sensors to provide fault detection and identification,
- Provides fault exclusion without the need to compute cross-filter covariances, and
- Provides a robust measure of system integrity without constraining the fault type [1]

Furthermore, the ARMAS framework has multiple operating modes, as listed in Figure 20, which allows sensor recovery through self-correcting methods. This work leverages the ARMAS framework to introduce an approach which utilizes multiple ARMAS subsystems to generate a navigation solution with a lower workload than a one ARMAS system instantiation.

Mode	Resilient sensor management objectives
Monitoring	Provide sensor-agnostic fault detection and exclusion
	Ensure fault-agnostic system integrity
	Trigger sensor model remedial measures
Validation	Initialize offline sensors without compromising integrity
	Validate questionable sensor models in real-time
	Provide independent verification of remedial measures
Calibration	Augment state-space with specified model parameters
	Follow a prescribed estimation sequence for observability
	Reduce state-space after specified termination criteria
Remodeling	Dynamically spawn multiple-model filter bank
	Select best candidate model using statistical criteria
	Delete filter bank after specified termination criteria

Figure 20: ARMAS operating modes [1]

4.4 System Implementation

This section details the implementation of a software system comprised of multiple ARMAS subsystems which henceforth will be referred to as the ARMAS subsystem model. The ARMAS subsystem model overview is first presented followed by the assumptions used in the engineering of this new model. Lastly, the software components that build the ARMAS subsystem model are discussed.

4.4.1 ARMAS Subsystem Model Overview

In this new implementation model, an ARMAS application wrapper encapsulates a top level controller, solution filters, and the ARMAS subsystems as shown in Figure 21. The original ARMAS system implementation model was a singular system architecture, as described in Chapter III. The layered approach detects and identifies sensor faults at the subsystem layer while processing the information at the top level to generate the most accurate solution to the user.

4.4.2 Applicable Scenario

The ARMAS system developed for this study is designed to support a probable operational scenario. In this type operational scenario, a vehicle equipped with the ARMAS subsystem model has access to multiple GNSSs relaying geo-spatial positioning data, along with an Inertial Navigation System (INS) to incorporate into an all-source navigation solution. In this paper, a satellite constellation is defined as GNSS satellites operating under the same frequency band. Therefore, in a GNSS degraded or denied environment, it would be probable that entire satellite constellation would be unavailable to the user, as opposed to loosing single satellites. By modeling GNSS satellites as ARMAS system sensors, this rationale can used to eliminate batches of sensors that respresent satellite constellations from the navigation solution

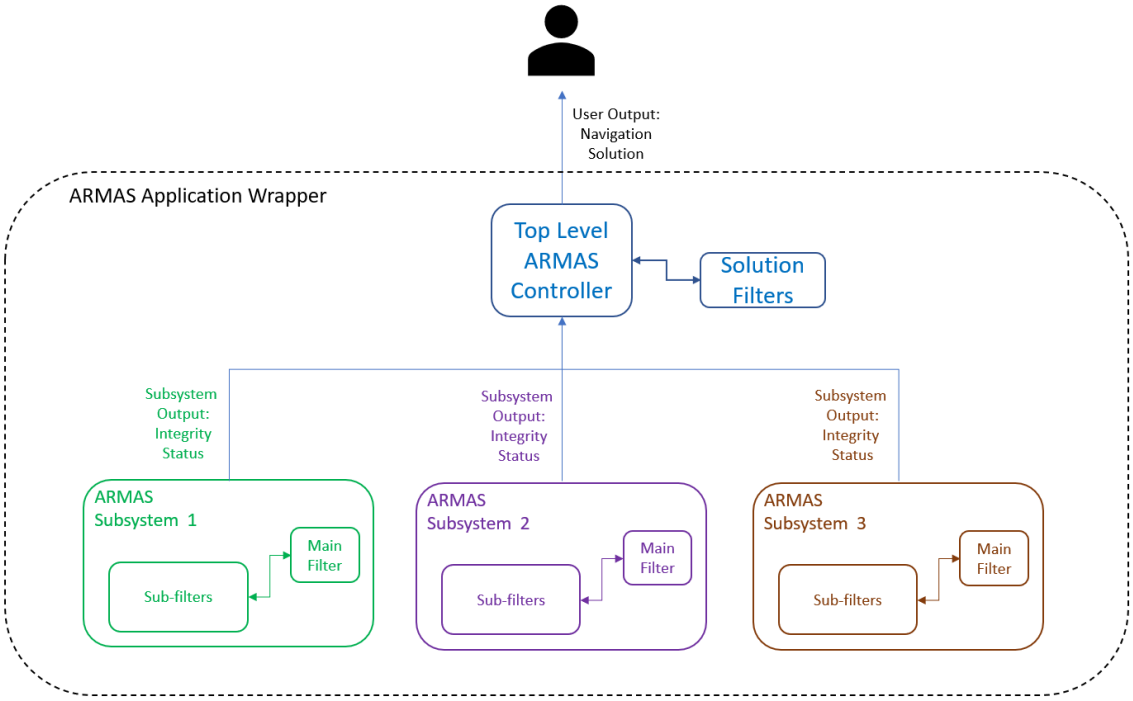

Figure 21: ARMAS subsystem model diagram.

while only detecting the initial fault(s) of the sensor batch. This approach reduces the size of the Kalman filter array, thus also reducing the computation workload of the system. For this test concept, the sensor fault detection was limited to one fault for clarity of presentation, however the system would not be limited in this way could be increased given the available computational power. This GNSS scenario is utilized as an interesting application to illustrate this novel capability, but this approach can be applied to any ARMAS system that can segregate sensors into batches/groups.

4.4.3 Assumptions

This study takes into account the following assumptions: First, it is assumed that ARMAS system sensors can be categorized into groups. Second, if one sensor in a group experiences a sensor fault, then it is highly probable that the other sensors in the same group will experience the same type of fault. Furthermore, this study assumes

at least one sensor group is available to provide accurate navigational data to the user at any given time. Lastly, ARMAS relies on the assumption that measurements provide full state observability.

4.4.3.1 ARMAS subsystem

Each ARMAS subsystem operates autonomously with a different arrangement of sensors and maintains an awareness of its own navigation integrity. This feature is important as the top level controller does not have sensor fault detection capabilities and instead relies on the information provided by the ARMAS subsystems to excluded sensors from the navigation solution.

Each ARMAS subsystem instantiation is comprised of a sensor group. For this test concept, a sensor group is composed of GNSS satellites that operate in the same frequency band and from the same constellation. For example, GPS satellites would require three ARMAS subsystems to accommodate the L1 (1575 MHz), L2 (1227 MHz), and L5 (1176 MHz) frequency bands. Each ARMAS subsystem may also include additional on-board vehicle sensors that provide alternative navigation inputs. For the simulated scenario, each ARMAS subsystem included one three-dimensional position sensor (e.g. as from a vision-map-matching navigation system for example) and one three-dimensional velocity sensor in addition to the satellite pseudoranges to demonstrate this capability.

Given the likelihood that adversary jamming and/or spoofing capabilities would affect all satellites transmitting in the targeted frequency band, identifying all simultaneous faults at a time instance would become computationally expensive redundant information. Minimizing the computational burden associated with detecting and identifying multiple simultaneous faults for this scenario, each subsystem is configured to only detect and identify one sensor fault as shown in Figure 22, where each

filter is missing a unique sensor. Future applications can tune the fault detection configuration to accommodate specific resources. If more than one sensor fault occurs in an ARMAS subsystem, the integrity of the subsystem has failed, raising an alarm flag. The flag is only lowered when all sensors fully recover and zero sensor measurement errors are detected. The status of this integrity flag is the output by each subsystem to the top level ARMAS controller.

4.4.3.2 Top Level Control Logic

The top level ARMAS controller is designed to output the most accurate navigation solution to the user. Taking into account that each ARMAS subsystem is reporting integrity of their respective sensors, the top level controller monitors these outputs for any subsystem failures. A set of solution filters account for the different subsystem failure possibilities. From the set of solution filters, the top level controller selects a Kalman filter that excludes the sensors associated with the faulty subsystem which generates the navigation solution to the user. This process allows the navigation solution to be generated from an uncorrupted filter regardless of subsystem failure under the assumption that there will always be at least one ARMAS subsystem online.

For example, using an ARMAS application with three subsystem as shown in Figure 21, the set of solution filters would consist of the seven filters as presented in Figure 23. In this example, Kalman filter 1 (SKF1) is used when all subsystem are running normally, Kalman filter 2 (SKF2) would be used when ARMAS subsystem three is down, Kalman filter 3 (SKF3) would be used when ARMAS subsystem two is down, and so on.

ARMAS Subsystem 1, Simulating frequency 1											
	Kalman filter array										
Sensor	Main	KF1	KF2	KF3	KF4	KF5	KF6	KF7	KF8	KF9	KF10
1	•		•	•	•	•	•	•	•	•	•
2	•	•		•	•	•	•	•	•	•	•
3	•	•	•		•	•	•	•	•	•	•
4	•	•	•	•		•	•	•	•	•	•
5	•	•	•	•	•		•	•	•	•	•
6	•	•	•	•	•	•		•	•	•	•
7	•	•	•	•	•	•	•		•	•	•
8	•	•	•	•	•	•	•	•		•	•
9	•	•	•	•	•	•	•	•	•		•
10	•	•	•	•	•	•	•	•	•	•	

ARMAS Subsystem 2, Simulating frequency 2											
	Kalman filter array										
Sensor	Main	KF1	KF2	KF3	KF4	KF5	KF6	KF7	KF8	KF9	KF10
1	•		•	•	•	•	•	•	•	•	•
2	•	•		•	•	•	•	•	•	•	•
3	•	•	•		•	•	•	•	•	•	•
4	•	•	•	•		•	•	•	•	•	•
5	•	•	•	•	•		•	•	•	•	•
6	•	•	•	•	•	•		•	•	•	•
7	•	•	•	•	•	•	•		•	•	•
8	•	•	•	•	•	•	•	•		•	•
9	•	•	•	•	•	•	•	•	•		•
10	•	•	•	•	•	•	•	•	•	•	

ARMAS Subsystem 3, Simulating frequency 3											
	Kalman filter array										
Sensor	Main	KF1	KF2	KF3	KF4	KF5	KF6	KF7	KF8	KF9	KF10
1	•		•	•	•	•	•	•	•	•	•
2	•	•		•	•	•	•	•	•	•	•
3	•	•	•		•	•	•	•	•	•	•
4	•	•	•	•		•	•	•	•	•	•
5	•	•	•	•	•		•	•	•	•	•
6	•	•	•	•	•	•		•	•	•	•
7	•	•	•	•	•	•	•		•	•	•
8	•	•	•	•	•	•	•	•		•	•
9	•	•	•	•	•	•	•	•	•		•
10	•	•	•	•	•	•	•	•	•	•	

Figure 22: ARMAS subsystem configuration for test scenario described in section 4.5, where Sensor 1 is a position sensor, sensor 2-9 are pseudorange sensors, and sensor 10 is a velocity sensor.

	Solution Filters						
ARMAS Subsystem	SKF1	SKF2	SKF3	SKF4	SKF5	SKF6	SKF7
1	•	•	•		•		
2	•	•		•		•	
3	•		•	•			•

Figure 23: Solution Kalman filters for an ARMAS application with three subsystems

4.5 Simulated Scenario

A simulation scenario was performed to investigate the behavior of the ARMAS subsystem model. The scenario was set to simulate three frequency bands being spoofed at different time intervals. This required three ARMAS subsystems running autonomously while the top level controller outputs the most accurate navigation solution to the user. The top level controller used a set of solution filters configured identically to arrangement shown in Figure 23.

To model a spoofing scenario, pseudorange sensor measurements were altered with an added distance bias. Spoofing the different ARMAS subsystem at different time intervals, as shown in Figure 24, examines the ability of the overall application to intelligently switch from a corrupt navigation solution over to an accurate solution. Additionally, it tests the autonomy of the entire system down to the subsystem levels. The desired output solution for the spoofing scheme shown in Figure 24 should be a vehicle flight path that closely follows the true flight given that there is always a batch of sensors in the scenario that are not being spoofed.

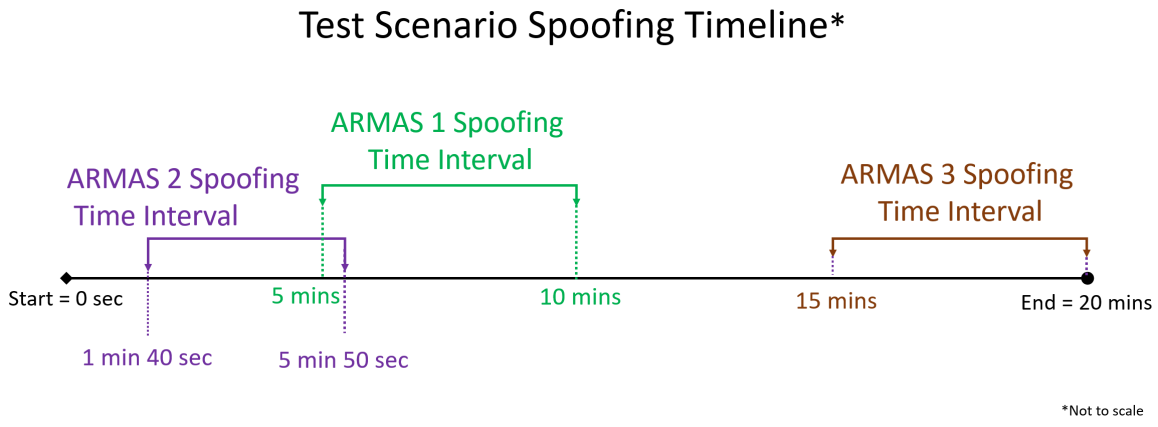

Figure 24: Test scenario timeline

The navigation solution for the test scenario was the vector $\dot{\mathbf{x}}(\mathbf{t})$, a ten system

state vector defined as

$$\dot{\mathbf{x}}(t) \;= \left[\dot{x_{p_x}}(t)\; \dot{x_{p_y}}(t)\; \dot{x_{p_z}}(t)\; \dot{x_{v_x}}(t)\; \dot{x_{v_y}}(t)\; \dot{x_{v_z}}(t)\; \dot{x_{a_x}}(t)\; \dot{x_{a_y}}(t)\; \dot{x_{a_z}}(t)\; \dot{x_c}(t) \right]^T, \quad (81)$$

where x_p is the vehicle's three-dimensional position [m], x_v is the velocity [m/s] for the three-dimensions, x_a is the acceleration [m/s^2] for the three-dimensions, and x_c is a clock error bias.

The initial position state estimates were set to zero for the x and y direction and 200 [m] in the z direction. The initial velocities were 4.25 [m/s] for the x direction, 5.03 [m/s] for the y direction and zero [m/s] for the z direction. These values were randomly generated for the first test scenario and used for the subsequent test runs. The initial acceleration state estimates were set to zero. The initial clock error bias was also randomly generated for the first test scenario and again, used for the subsequent test runs. The initial randomly generated clock error bias was (4.4083×10^3) [m].

The initial state estimation error covariance was set to 10^2 [m^2] in position, 10^2 [m^2/s^2] in velocity, $(1\times10^{-2})^2$ [m^2/s^4] in acceleration, and 8000^2 [m^2] for the clock error bias. Each trial was propagated using $\Delta t_k = 0.5$ [s], starting at $t_{k_{start}} = 0$ [s] and ending at $t_{k_{end}} = 1200$ [s], with induced faults continuously occurring during the spoofing time intervals shown in Figure 24. The false alarm test significance level was set to $\alpha = 1/500000 = 2.0 \times 10^{-6}$, while the test epoch was set to M = 2400 samples, which is equivalent to 1200 [s], for all test experiments.

Each ARMAS subsystem used sensor models inherited from the original ARMAS implementation [1] that included one three-dimensional position sensor identified as Sensor 1 modeled by,

$$\mathbf{z_k^{[1]}} \;=\; \mathbf{s} \odot \mathbf{x_{p_k}} \;+\; \mathbf{v_k^{[1]}}, \quad (82)$$

$$\mathbf{v_k^{[1]}} \sim \mathcal{N}\left(\underset{\mathbf{3x1}}{\mathbf{0}}, \mathbf{100^2}\,\underset{\mathbf{3x3}}{\mathbf{I}}\right), \tag{83}$$

where $s = [s_x\ s_y\ s_z]^T$ is a three-dimensional scale factor, $\odot$ symbolizes the Hadamard product, and I is an identity matrix. Sensors 2 - 9 are pseudorange sensors modeled by,

$$\mathbf{z_k^{[2...9]}} = |\mathbf{t_{[2...9]}} - \mathbf{x_{p_k}}| + \mathbf{b_k} + \mathbf{v_k^{[2...9]}}, \tag{84}$$

$$\mathbf{v_k^{[2...9]}} \sim \mathcal{N}\left(\mathbf{0}, \mathbf{10^2}\right), \tag{85}$$

where t is a three-dimensional position of the satellite, x_{p_k} is the three-dimensional position of the vehicle at t_k and b_k is a First Order Gauss-Markov process simulating a simple receiver clock error with a time constant $\tau = 3600\ [s]$ and $\sigma^2 = 8000^2\ [m^2]$.

Lastly, sensor 10 is a three-dimensional velocity sensor modeled by,

$$\mathbf{z_k^{[10]}} = \mathbf{s} \odot \mathbf{x_{v_k}} + \mathbf{v_k^{[10]}}, \tag{86}$$

$$\mathbf{v_k^{[10]}} \sim \mathcal{N}\left(\underset{\mathbf{3x1}}{\mathbf{0}}, \mathbf{50^2}\,\underset{\mathbf{3x3}}{\mathbf{I}}\right). \tag{87}$$

The described sensors correspond to the sensor arrangements in Figure 22.

Each subsystem was configured to only detect one sensor fault. This required 10 Kalman filters that need to be maintained by each ARMAS subsystem. Alternatively, single ARMAS system architecture, as described in section 4.2.3, would require 2,533,987 Kalman filters to be maintain to exclude one subsystem set of faulty pseudorange sensor (8 sensors) from the navigation solution.

The ARMAS subsystem model test scenario was initiated with warm up cycles to optimize and stabilize memory operations. After the warm up, 20 iterations of the test case was recorded for execution time and the results were averaged for a final value as shown in Figure 25. The iteration max time was the longest execution time taken to complete one Δt_k worth of data. This data point was used to analyze

whether the ARMAS subsystem model could execute under the real-time constraint of reading sensor measurement data every .5 seconds.

Test Run	Total Execution Time (secs)	Iteration Max Time (secs)
1	60.263	0.060211
2	60.933	0.058542
3	61.265	0.059107
4	60.538	0.060361
5	60.249	0.055349
6	60.685	0.058852
7	60.228	0.058571
8	60.204	0.056852
9	60.812	0.058587
10	60.417	0.056397
11	60.215	0.060479
12	60.083	0.056283
13	60.028	0.056713
14	59.716	0.056438
15	60.216	0.058788
16	60.018	0.051816
17	60.548	0.054985
18	60.578	0.059081
19	60.928	0.051652
20	60.634	0.059664
Avg	60.428	0.057436

Figure 25: ARMAS subsystem model test scenario run times

4.6 Results

The test scenario using the ARMAS subsystem model exhibited encouraging results. The navigation solution output by the overall system follows the true path with slight deviations due to the simulated measurement process noise as shown in Figure 29. The ARMAS subsystems were able to function autonomously and self identify when the integrity of the subsystem failed. More importantly, when faulty sensors in the subsystems recovered, the subsystems were able to ascertain when its navigation solution was free from corrupt measurements.

Figures 26, 27, and 28, show the navigation solution output of each subsystem. As each subsystem experienced more than 1 sensor fault, the subsystem is unable to identify the faulty sensors due to the limited size of the Kalman filter array and therefore unable to remove them from the navigation solution output, however, each ARMAS instance is still able to determine that there is an integrity problem with the solution and "self-eliminate". The integrity alarm flag gets raised alerting the top level controller of the failure occurring in the subsystem. The corrupt sensor measurements cause the navigation to deviate from the true flight path. Even though the solution is excluded from the final solution, each ARMAS instance continues to attempt to recover.

The top level controller functioned as designed by excluding the faulty sensors associated with the fail subsystem and generating navigation solution from an uncorrupt filter, maintaining navigation integrity at a much lower computational cost than the original ARMAS design would require. The final output to the user shows a navigation solution without significant divergence from the true flight path.

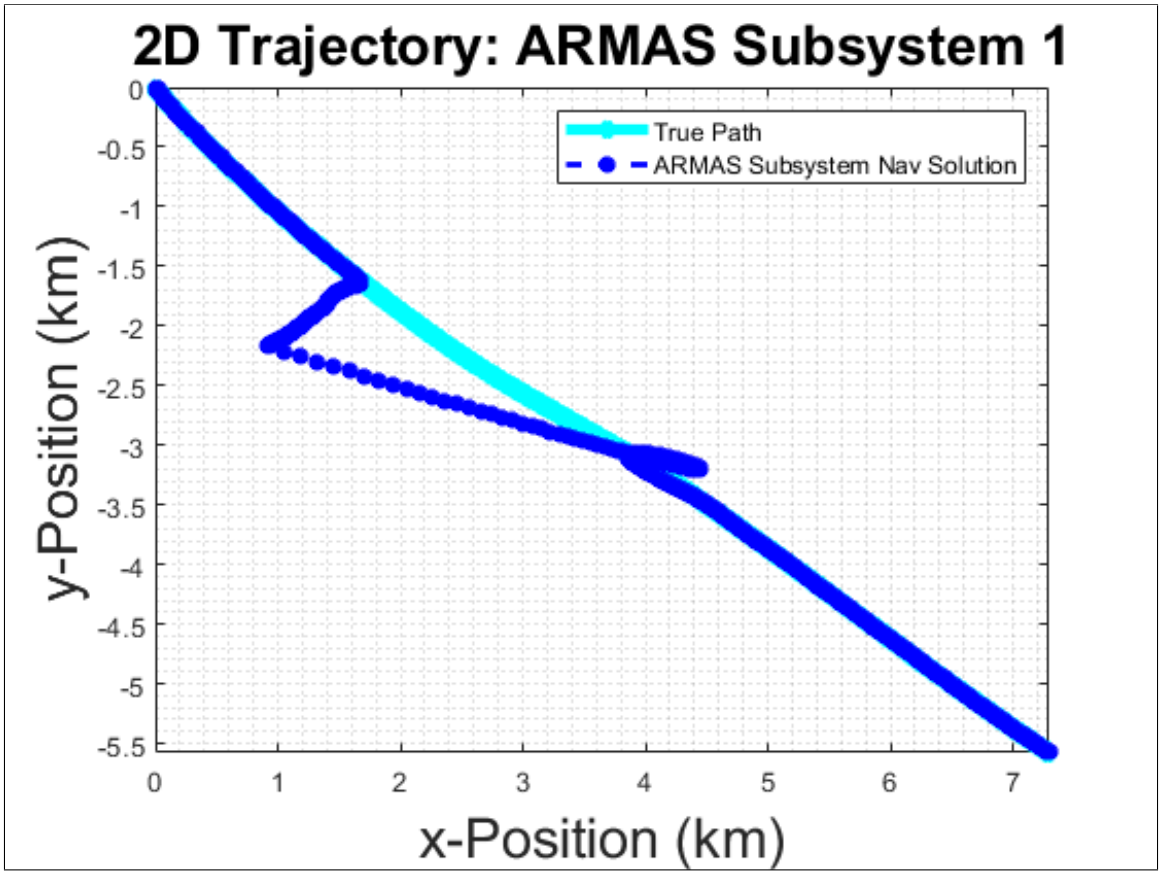

Figure 26: ARMAS Subsystem 1 Navigation Solution

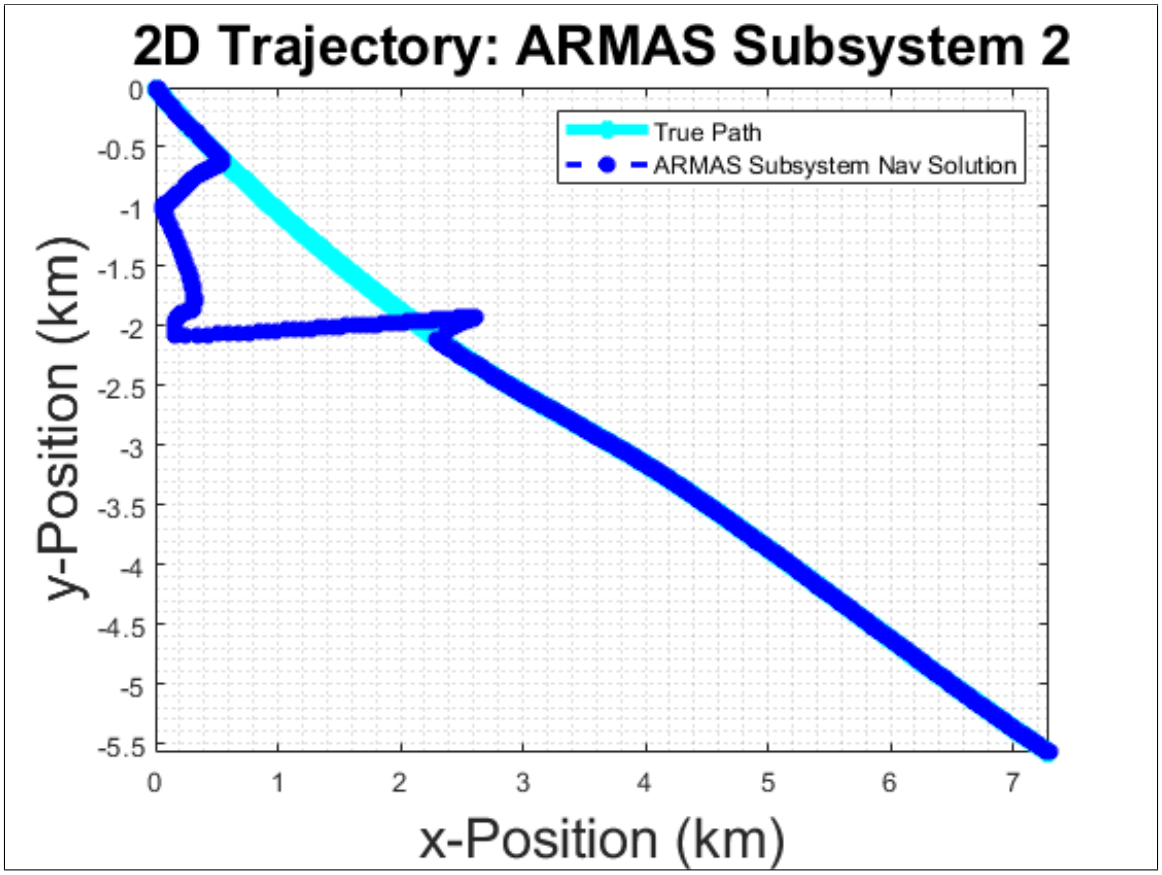

Figure 27: ARMAS Subsystem 2 Navigation Solution

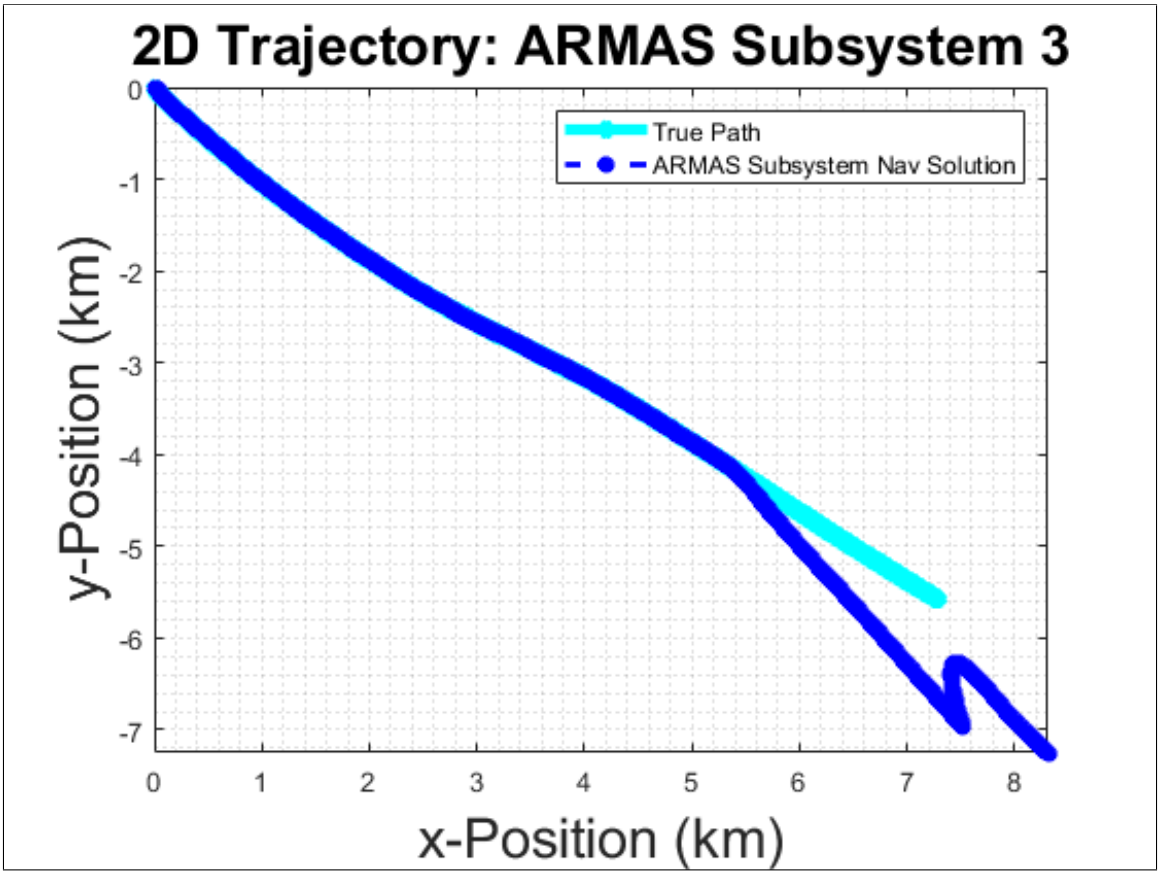

Figure 28: ARMAS Subsystem 3 Navigation Solution

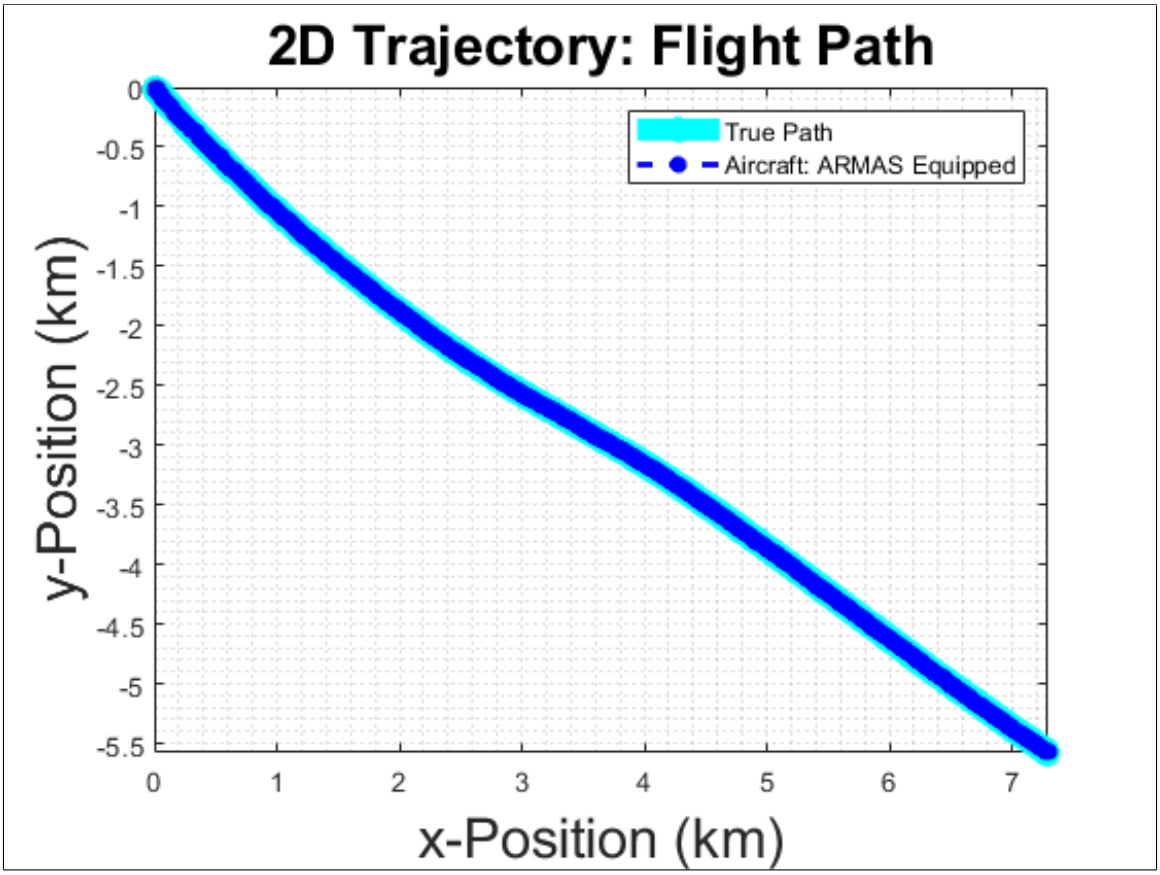

Figure 29: ARMAS Final Navigation Solution

4.7 Future Works

The ability to fully characterize the behavior of an ARMAS subsystem model depends largely on the test cases perform. Subjecting the new ARMAS model discussed in this paper to a wider range of scenarios would provide better insight into the strengths and weaknesses of the approach. Enhancing the experiments with real-world test data would assess the ARMAS subsystem for operational suitability. Additionally, it would allow a comparison between experimental results and existing navigation outputs to validate the benefits, if any, for the end user.

4.8 Conclusion

This research presented a novel approach of multiple instantiations of ARMAS to obtain a navigation solution with integrity that is robust to multiple simultaneous sensor failures of GNSS pseudoranges. The approach demonstrated that running 37 simultaneous filters provided equivalent performance to the 2,533,987 simultaneous filters that would be required under the original formulation of ARMAS. This study provided an initial proof of concept for a modified ARMAS implementation model under the assumptions outline in section 4.4.3. The information in this paper leads to the recognition that operational situations may alleviate the computational burden required to produce a practical navigation solution given expected sensor fault behavior. Results show promise that a real-time ARMAS system can be achieved for large scale sensor applications through the segregation of system sensors by functional characteristics and fusing results from multiple ARMAS subsystem into one navigation solution.

V. Conclusions

This chapter consolidates the conclusions discussed in Chapter III and Chapter IV respectively into an overall evaluation focused on answering the research question presenting in Chapter I.

5.1 Conclusions of Research

The over-arching investigative question was: How can the academic ARMAS software be optimized towards an operational real-time system?

Chapter I hypothesized that a parallel processing software model implemented on a GPU device could optimize the overall execution time of a program application after an ARMAS system sensor size and detectable fault configuration reaches a substantial quantity, otherwise a CPU sequentially processing model would produce superior results. Additionally, it was proposed that a single ARMAS system instantiation is less computationally efficient than an association of smaller ARMAS systems working collectively to produce the same output under a set bounded conditions.

Chapter III presented a software model that used parallel processing on a GPU to accelerate program execution time. The approach parallelized Kalman filter instances and other computation tasks to improve the ARMAS framework performance. Test cases incrementally scaled up the system sensor size and detectable system faults to evaluate computational load and measure execution time. Small sensor size configurations proved to be faster on a CPU, while a 3.5 order of magnitude speedup was achieved by the GPU ARMAS implementation over the CPU version on a large sensor system configuration. Therefore, the results show the current academic ARMAS software can optimize the execution of large scale ARMAS applications through parallel processing.

Chapter IV presented a novel approach that utilized ARMAS subsystems to obtain a navigation solution. An initial test case scenario demonstrated an ARMAS subsystem model could output an accurate navigation solution while reducing the computation demand when compared to a single ARMAS system architecture. This study provided an initial proof of concept for a modified ARMAS implementation model under the assumptions outline in section 4.4.3. Recognising that operational situations may alleviate the computational burden required to produce a practical navigation solution given an expected behavior, the ARMAS subsystem model can optimize computational resources through selective fault detection.

Overall results show promise that a real-time ARMAS system can be achieved for operational navigation systems in the near future with the optimization method discussed in this thesis and continued advancements in hardware technology.

5.2 Future Work

Throughout this research, the following key areas were identified as potential follow-on work.

- Implement an improved GPU eigen solver using the more efficient QR algorithm to decrease overall runtimes.
- Analyze a traditional heterogeneous computing model to suit workload with appropriate hardware
- Expand testing scenarios for the ARMAS subsystem model
- Develop a navigation fusion approach in the top layer controller for the ARMAS subsystem model

OPTIMIZING BANKS OF KALMAN FILTERS FOR NAVIGATION INTEGRITY USING PARALLEL COMPUTING AND EFFICIENT SOFTWARE DESIGN

A. C++/CUDA Code

Available upon request: luis.sepulveda@us.af.mil or robert.leishman@afit.edu

B. Accelerating the ARMAS framework Result Tables

CPU Execution Time (secs)			
Faults	1	2	3
Number of System Sensors			
10	11.53	24.48	63.10
14	34.17	72.47	206.50
17	55.10	113.51	486.69
21	86.85	225.37	1276.55
25	143.55	372.51	3192.92
29	231.71	629.25	7643.59
33	379.90	921.81	15938.54
36	452.21	1209.81	29751.12
40	572.48	1703.88	50935.92

GPU Execution Time (secs)			
Faults	1	2	3
Number of System Sensors			
10	51.66	111.18	117.55
14	99.77	173.48	211.82
17	144.93	213.19	495.48
21	203.02	282.02	1040.54
25	244.04	362.05	2152.26
29	317.82	455.58	3560.94
33	359.82	576.84	5906.38
36	415.20	703.16	8412.91
40	470.30	840.70	14425.10

Bibliography

1. Juan D. Jurado and John F. Raquet. Autonomous and Resilient Management of All-Source Sensors. *Proceedings of the ION 2019 Pacific PNT Meeting*, pages 142–159, 2019.

2. Bjarne Stroustrup. An overview of C++. *Proceedings of the 1986 SIGPLAN Workshop on Object-Oriented Programming, OOPWORK 1986*, (October):7–18, 1986.

3. Bjarne Stroustrup. A history of C++: 1979-1991. *2nd ACM SIGPLAN Conference on History of Programming Languages, HOPL 1993*, pages 271–297, 1993.

4. Wellson Giri, Shailendra Giri, and Prof Subarna Shakya. An Overview of C and C ++ : A Case of Engineering Faculty. pages 497–501, 2019.

5. Bjarne Stroustrup. *A Tour of Cpp*. Addison-Wesley Professional, 2018.

6. C Saravanan P Geetha. Qualified Study of C , C ++ and C # Programming Languages. (41):64–71, 2020.

7. Anastasia Kazakova. Infographic: C/C++ facts, 2015.

8. Bjarne Stroustrup and Morgan Stanley. Thriving in a Crowded and Changing World : C ++ 2006 – 2020. 4(June), 2020.

9. L Goldthwaite. Technical Report on C ++ Performance. *Iso-18015*, 2006:202, 2006.

10. Tyler Andrews. Computation Time Comparison Between Matlab and C++ Using Launch Windows. *Aerospace Engineering*, pages 1–6, 2012.

11. Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. *Technical Report 2000-5*, 2000.

12. Hrvoje Jasak, Aleksandar Jemcov, and Zeljko Tukovic. OpenFOAM : A C ++ Library for Complex Physics Simulations. *International Workshop on Coupled Methods in Numerical Dynamics*, m:1–20, 2007.

13. John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA C Programming*, volume 53. 2013.

14. David Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors*. 2013.

15. Grayson Fenwick and Cindy Norris. GPGPU programming for CS undergraduates: Which one is superman? *ACMSE 2020 - Proceedings of the 2020 ACM Southeast Conference*, pages 2–9, 2020.

16. Anantha P. Chandrakasan, Miodrag Potkonjak, Renu Mehra, Jan Rabaey, and Robert W. Brodersen. Optimizing power using transformations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(1):12–31, 1995.

17. Aaftab Munshi, Benedict Gaster, Timothy Mattson, James Fung, and Dan Ginsburg. *the Opencl Programming Guide*. Pearson Education, Inc., 2012.

18. Marius Bitca, Darinela Andronovici, and Dumitras Mamaliga. CONCURRENCY AND PARALLELISM , BETWEEN PROGRAMMING AND REAL LIFE. pages 1–3, 2020.

19. Hong Zhang, Da-Fang Zhang and Xia-An Bi. Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU. 2012.

20. Sparsh Mittal and Jeffrey S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, 47(4), 2015.

21. Mario A.M. Chapa and Sato Hiroyuki. A linear performance-breakdown model for GPU programming optimization guidance. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*, pages 596–603. IEEE Computer Society, nov 2014.

22. R. E. Kalman. A New Approach to Linear Filtering and Prediction Problems. Technical report, Research Institute for Advance Study, Baltimore, 1960.

23. Charles F Van Loan. Computing Integrals Involving the Matrix Exponential. *IEEE Transactions on Automatic Control*, 23(3):395–404, 1978.

24. Greg Welch and Gary Bishop. An Introduction to the Kalman Filter. Technical report, 2006.

25. Peter S Maybeck. *Stochastic Models, Estimation, and Control. Volume 1.* Navtech, Virginia, 1982.

26. Peter S Maybeck. *Stochastic Models, Estimation, and Control. Volume 2.* Navtech, Virginia, 1984.

27. Kyle J Kauffman. SCORPION. https://www.afit.edu/docs/Scorpion.pdf, 2021.

28. R. De Maesschalck, D. Jouan-Rimbaud, and D. L. Massart. The Mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems*, 50(1):1–18, 2000.

29. Min Yu Huang, Shih Chieh Wei, Bormin Huang, and Yang Lang Chang. Accelerating the Kalman filter on a GPU. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pages 1016–1020, 2011.

30. Juan Manuel Rodríguez, Francisco Gómez Fernández, María Elena Buemi, and Julio Jacobo-Berlles. Dynamic textures segmentation with GPU. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7441 LNCS(L):607–614, 2012.

31. Rafael Cisneros-Magana, Aurelio Medina, and Venkata Dinavahi. Parallel Kalman filter based time-domain harmonic state estimation. *45th North American Power Symposium, NAPS 2013*, 2013.

32. Francisco Gómez Fernández, María Elena Buemi, Juan Manuel Rodríguez, and Julio C. Jacobo-Berlles. Performance of dynamic texture segmentation using GPU. *Journal of Real-Time Image Processing*, 11(2):375–383, 2016.

33. Rafael Cisneros-Magaña, Aurelio Medina, Venkata Dinavahi, and Antonio Ramos-Paz. Time-Domain Power Quality State Estimation Based on Kalman Filter Using Parallel Computing on Graphics Processing Units. *IEEE Access*, 6:21152–21163, apr 2018.

34. Woochul Kang and Jibum Kim. PDDS: Scalable Sensor Data Distribution for Cyber-Physical Systems Using GPGPUs. *IEEE Internet of Things Journal*, 5(3):2025–2036, 2018.

35. Maozhen Li, Institute of Electrical and Electronics Engineers, and IEEE Circuits and Systems Society. *Optimization of Parallel Algorithm for Kalman Filter on CPU-GPU Heterogeneous System.* 2016.

36. Olov Rosén and Alexander Medvedev. Efficient Parallel Implementation of a Kalman Filter for Single Output Systems on Multicore Computational Platforms. Technical report, 2011.

37. O. Rosén, A. Medvedev, and T. Wigren. Parallelization of the Kalman filter on multicore computational platforms. *Control Engineering Practice*, 21(9):1188–1194, sep 2013.

38. Myint Myint Thein and Pho Kaung. *Parrallel MPI/C++ Programming for the Kalman Filter*. 2013.

39. Hadis Karimipour and Venkata Dinavahi. Extended Kalman Filter-Based Parallel Dynamic State Estimation. *IEEE Transactions on Smart Grid*, 6(3):1539–1549, 2015.

40. Abdul Lateef Yussiff, Suet Peng Yong, and Baharum B. Baharudin. Parallel Kalman filter-based multi-human tracking in surveillance video. *2014 International Conference on Computer and Information Sciences, ICCOINS 2014 - A Conference of World Engineering, Science and Technology Congress, ESTCON 2014 - Proceedings*, 2014.

41. Abdenour Amamra and Nabil Aouf. GPU-based real-time RGBD data filtering. *Journal of Real-Time Image Processing*, 14(2):323–340, 2018.

42. John F. Raquet Kenneth A. Fisher. Precision Position, Navigation, and Timing without the Global Positioning System. *Air and Space Power Journal*, 25:24–33, 2011.

43. Gaël Guennebaud, Benoît Jacob, and Others. Eigen v3. http://eigen.tuxfamily.org, 2010.

44. Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*, pages 167–188. Springer, 2014.

45. John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

46. John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

47. Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, and Enrique S. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. *IPDPS Miami 2008 - Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium, Program and CD-ROM*, 2008.

48. John Burkardt. JACOBI_EIGENVALUE, Eigenvalues and Eigenvectors of a Symmetric Matrix.

49. Frank Van Graas and James L Farrell. Baseline fault detection and exclusion algorithm. In *Proceedings of the 49th Annual Meeting of The Institute of Navigation (1993)*, pages 413–420, 1993.

50. Bradford W Parkinson and Penina Axelrad. Autonomous GPS integrity monitoring using the pseudorange residual. *Navigation*, 35(2):255–274, 1988.

51. Mark A Sturza. Navigation system integrity monitoring using redundant measurements. *Navigation*, 35(4):483–501, 1988.

52. Mats Brenner. Implementation of a RAIM Monitor in a GPS Receiver and an Integrated GPS/IRS. In *Proceedings of the 3rd International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GPS 1990)*, pages 397–414, 1990.

53. Mats Brenner. Integrated GPS/inertial fault detection availability. *Navigation*, 43(2):111–130, 1996.

54. B Brumback and M Srinath. A chi-square test for fault-detection in Kalman filters. *IEEE Transactions on Automatic Control*, 32(6):552–554, 1987.

55. Curt Call, Mike Ibis, Jim McDonald, and Kevin Vanderwerf. Performance of Honeywell's inertial/GPS hybrid (high) for RNP operations. In *Position, Location, and Navigation Symposium*, pages 25–27, 2006.

56. T Kerr. Statistical analysis of a two-ellipsoid overlap test for real-time failure detection. *IEEE Transactions on Automatic Control*, 25(4):762–773, 1980.

57. Ryan S Y Young, Gary A McGraw, and Brian T Driscoll. Investigation and comparison of horizontal protection level and horizontal uncertainty level in FDE algorithms. In *Proceedings of the 9th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GPS 1996)*, pages 1607–1614, 1996.

58. Ryan S Y Young and Gary A McGraw. Fault detection and exclusion using normalized solution separation and residual monitoring methods. *Navigation*, 50(3):151–169, 2003.

59. Umar Iqbal Bhatti. An improved sensor level integrity algorithm for GPS/INS integrated system. In *Proceedings of the 19th International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS 2006)*, pages 3012–3023, 2006.

60. Umar I Bhatti, Washington Y Ochieng, and Shaojun Feng. Integrity of an integrated GPS/INS system in the presence of slowly growing errors. Part II: analysis. *GPS Solutions*, 11(3):183–192, 2007.

61. Umar I Bhatti, Washington Y Ochieng, and Shaojun Feng. Integrity of an integrated GPS/INS system in the presence of slowly growing errors. Part I: A critical review. *Gps Solutions*, 11(3):173–181, 2007.

62. Shih-Yih R Young and Gary A McGraw. Method and system for fault detection and exclusion for multi-sensor navigation systems, may 2007.

63. Ren Da and Ching-Fang Lin. Sensor failure detection with a bank of kalman filters. In *Proceedings of 1995 American Control Conference-ACC'95*, volume 2, pages 1122–1126. IEEE, 1995.

64. Mathieu Joerger and Boris Pervan. Kalman filter-based integrity monitoring against sensor faults. *Journal of Guidance, Control, and Dynamics*, 36(2):349–361, 2013.

65. John D. Quartararo and Steven E. Langel. Detecting Slowly Accumulating Faults Using a Bank of Cumulative Innovations Monitors in Kalman Filters. *Proceedings of the 33rd International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+ 2020)*, (20):2367–2381, 2020.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
26–03–2021	Master's Thesis	Jan 2020 — Mar 2021

4. TITLE AND SUBTITLE

Optimizing Banks of Kalman Filters for Navigation Integrity using Parallel Computing and Efficient Software Design

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Sepulveda, Luis E., Capt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-21-M-079

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Alternative navigation is an area of research which employs a variety of sensor technologies to provide a navigation solution in Global Navigation Satellite System degraded or denied environments. The Autonomy and Navigation Technology Center at the Air Force Institute of Technology has recently developed the Autonomous and Resilient Management of All-source Sensors (ARMAS) navigation framework which utilizes an array of Kalman Filters to provide a navigation solution resilient to sensor failures. The Kalman Filter array size increases exponentially as system sensors and detectable faults are scaled up, which in turn increases the computational power required to run ARMAS in a real-world application. In an effort to engineer a real-time ARMAS system, this study developed C++ CPU and GPU versions to examine the performance trade-offs as system sensors and detectable faults are scaled up. Results show promise that a real-time ARMAS system can be achieved for large scale applications through parallel processing on a many-core processor architecture.

15. SUBJECT TERMS

General-Purpose Graphics Processing Unit (GPGPU), parallel computing, Kalman filter, software engineering, optimization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Robert C. Leishman, AFIT/ENG
U	U	U	UU	84	19b. TELEPHONE NUMBER *(include area code)* (937)255-3636 x4755; Robert.Leishman@afit.edu