Hardware Acceleration for Pedestrian Dead Reckoning in Embedded Systems

by

Benjimen Tucker Johnston

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 7, 2022

Keywords: Hardware, Acceleration, GPU, PDR, Kalman, Navigation, IMU

Approved by

Christopher Harris, Chair, Assistant Professor of Electrical and Computer Engineering
Spencer K. Millican, Assistant Professor of Electrical and Computer Engineering
John Y. Hung, Professor of Electrical and Computer Engineering

Abstract

Hardware acceleration within embedded systems can potentially allow algorithms to meet real time requirements in devices where it was previously impossible. Though many algorithms have been developed targeting embedded systems, their ability to meet the target environment's real time demands is often unclear.

The primary focus of this thesis is the effects of GPU acceleration on a pedestrian dead reckoning system (PDR) targeting processing environments representative of the current wearable market. A software PDR system was developed from components used popularly within research. The system was validated by evaluating accuracy of results from each component against ground truth data. The validated system was simulated on several processors to generate a set of baseline execution times relative to the host processor. The most computationally intense component was selected based on profiling results and accelerated using a GPU. The computational speedup of the system was then used to determine expected execution time relative to each processor baseline execution time.

This research was the first to apply hardware acceleration techniques to an embedded PDR system. The results showed a 86% speedup of the accelerated system's CPU execution but no decrease in overall system computation time. Additionally, the benefits of this change were shown to allow certain particularly low performance processors to meet real time requirements. This shows that GPU acceleration can be applied to accelerate embedded algorithms to allow for smaller and cheaper processing systems to be used compared to those used previously.

Acknowledgments

I would like to thank my advisor and committee chair, Dr. Christopher Harris, for his guidance while I was a graduate student. His help was invaluable in the completion of this work.

I would like to thank Dr. John Hung and Dr. Spencer Millican for their support and evaluation of my work.

I would like to thank Paul Atilola for his support as a sounding board for solving issues in the development of this system.

I am thankful for the support of my family during this work and particularly during the final weeks of study.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Within embedded systems, there are some algorithms which would be useful to run in a low resource environment but are too computationally intensive. They cannot meet the requirements to run real time or drain power too quickly to be of substantial use in a practical implementation. This is particularly true in the case of wearable systems which, ideally, the user can forget they are wearing due to the small size and weight. Hardware acceleration techniques can be applied to these algorithms to both distribute computational intensity and reduce power consumption to meet the real time requirements and decrease power consumption. This work investigates the application of GPU acceleration to one such case to improve execution time of algorithms for personal tracking of an individual.

Global navigation satellite systems (GNSS) are a technology which has allowed for accurate positioning services on a global scale since the United States Department of Defense gave public access to its GPS system in 1983. This and other GNSS have given rise to popular use of location-based services (LBS) in a plethora of commercial applications. LBS are applied in areas from car navigation systems to mobile video gaming and each area comes with its own additional challenges. In the case of systems which are related to the tracking of an individual person, terrain and buildings often interfere with or completely block communication with GNSS. GNSS are then often frequently inadequate for precise tracking of an individual.

This is where pedestrian dead reckoning (PDR) can be of use. PDR aims to provide accurate positional tracking of an individual in cases where GNSS signals become inaccurate or useless. This is accomplished by using an initial trusted reference position from GNSS or some other external source and extrapolating position using only on-board sensors. When a

system's GNSS signal becomes unreliable, a PDR system could take over until such a time as the connection is reestablished.

PDR, however, is far from a solved problem. Many different approaches have been proposed that rely on technologies like neural networks[1][10], sensor signal analysis[6], and Kalman filtering[2][4]. While there are systems which achieve a high degree of accuracy, there is little evidence they can run in real time[13] in an embedded environment. Conversely, systems which can run in real time show a lower degree of accuracy[13] or are not complete PDR systems[12].

The technologies relied upon by these high accuracy systems generally carry a high degree of hardware exploitable parallelism. Therefore, hardware acceleration techniques can be applied to improve the performance of these systems and potentially allow them to operate in real time. While neural networks[14] and signal analysis have been accelerated many different ways, including with field programmable gate arrays (FPGAs)[16][17] and GPUs[14][15], the various small dimension Kalman filters have not received much attention for acceleration. Though the techniques used to perform signal analysis on input signals tend to show high degrees of parallelism, each approach generally looks substantially different from a programming perspective. This prevents hardware acceleration that can be easily used by a variety of approaches. This leaves Kalman filtering as a viable candidate for acceleration as its different variations are used similarly in many different approaches and their acceleration has not been investigated substantially.

The remaining chapters of this thesis are arranged as follows. Chapter 2 covers background information relating to the algorithm accelerated, GPU acceleration for our target architecture, and the profiling tools used. Chapter 3 discusses the standard design of a PDR system and the somewhat standard approaches used for this work. Chapter 4 describes the initial data collected and the resulting acceleration work. Chapter 5 describes the results of the acceleration and simulations of selected processors. Chapter 6 analyzes the results and summarizes the accomplishments of this work.

Chapter 2

Background

This chapter begins by covering the theory behind the algorithms selected for acceleration. It then covers computer architecture information related to accelerating CPU code with a GPU. The chapter concludes with a description of the profiling tools used by this work.

2.1 Review of the Kalman Filter

In this work, the Kalman filter is used to combine separate sets of measurements to produce a more accurate reading in a process referred to as sensor fusion. This is done in such a way that the complimentary features of each measurement source can offset the other's shortcomings. For example, accelerometer data is unstable from reading to reading but its average over time is highly accurate. On the other hand, a gyroscope's readings are precise from reading to reading but exhibit a large drift over time. The Kalman filter allows them to be combined such that the result has the gyroscope's moment to moment stability and the accelerometer's stability over time but little of either's instability.

The Kalman filter is a mean square optimal linear estimation algorithm that combines multiple measurements to produce an estimate of system state. This algorithm depends on a complete model of the system dynamics, or its estimate will have high error. The filter minimizes the mean square error of the estimation in the presence of Gaussian noise. The estimates tend to be closer to the true state than can be obtained from any measurement alone. In other words, the Kalman filter combines the measurements from multiple sources to produce an estimate of a system's state. This estimate is more accurate than one that can be derived from any measurement alone.

The discrete-time implementation of the Kalman filter can be broken into the predict stage and update stage. The predict stage uses a model of the system dynamics, the previous state of the system, the error covariance of the previous state, and an optional control input to predict the next state and provide a prediction error covariance. These are formally referred to as *a priori* values and denoted with subscript $k|k-1$. The update stage uses the state prediction, covariance of the state prediction, and a measurement to produce an estimate of the true state and error covariance matrix. These are referred to as *a posteriori* values and denoted with subscript $k-1|k-1$. The Kalman estimate values fall between the value from its prediction and the value obtained from measurement. The noise associated with the system model and with the measurement values determines which value the estimate falls closer to. This allows for the tuning of the estimator to closer fit the true state values.

2.1.1 Predict Stage

The predict stage calculates the state prediction, $\hat{x}_{k|k-1}$, and the error covariance of the state prediction, $P_{k|k-1}$. The state prediction is calculated by finding how the state model predicts change from the previous true state. If there is a control input, it is weighted by the control-input matrix and added to the updated state. The equations for state prediction and error covariance are respectively:

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k \tag{2.1}$$

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k \tag{2.2}$$

here $\hat{x}$ and P describe the state and error covariance matrix. The matrix F_k describes the state-transition model of the system. Matrices B_k and u_k describe the control-input model and the control input respectively, and Q_k describes the covariance of the process noise or the noise associated with the control input.

As stated in section 2.1, the Kalman filter is commonly used to combine the measurements from multiple separate sources. This is normally accomplished by using one measurement as the control input u_k with an appropriately designed B matrix. Alternatively, the measurement be an input of F_k such that the state-transition model varies from step to step.

2.1.2 Update Stage

The update stage uses the measurement input z_k to correct the estimate according to the calculated Kalman gain matrix, K_k. The Kalman gain represents how the difference between prediction and measurement is weighted and controls how the error covariance matrix is updated. The observation space output of the system is obtained by multiplying the measurement model with the state

$$\hat{y}_k = H_k \hat{x}_{k|k} \tag{2.3}$$

where H_k represents the measurement model. The equations for the update stage are given as

$$K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} \tag{2.4}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k (z_k - H_k \hat{x}_{k|k-1}) \tag{2.5}$$

$$P_{k|k} = (I - K_k H_k) P_{k|k-1} \tag{2.6}$$

where K_k is the Kalman gain. The measurement model is represented by H_k and maps the state space into the observed space. The measurement noise associated with the measurement is R_k. The external input to the update stage is z_k which is the measurement in the observed space. Equation 2.4 is used to calculate the Kalman gain matrix. Equations 2.5 and 2.6 are for state update and error covariance update respectively.

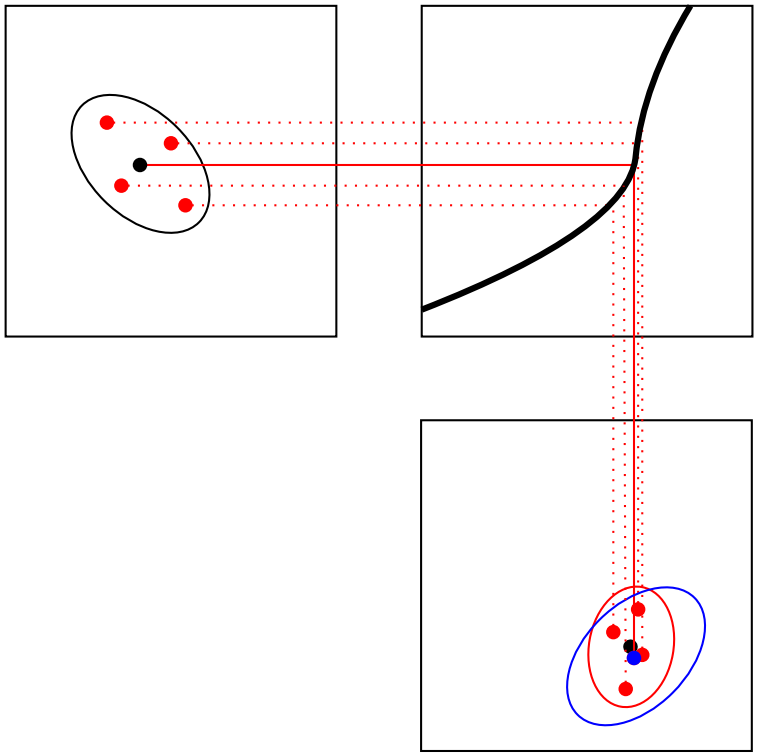

Figure 2.1: A model of how UKF's sigma points are used to obtain a more accurate result.

2.2 Unscented Transform of Kalman filter - Unscented Kalman Filter

While the Kalman filter performs well for systems with a linear or nearly linear system model and measurement model, it suffers from higher error with higher degrees of non-linearity. The unscented transform attempts to eliminate this accumulation of error.

Instead of propagating the state and error covariance through linearized models, the unscented transform picks a statistically representative set of sigma points spread around the state. These sigma points are selected such that the mean and covariance of the set is equal to the state and error covariance. Once propagated through the non-linear state-transition and measurement models, the mean of these sigma points is a more accurate representation of the true state.

Figure 2.1 is graphical representation of the unscented transform's impact where the top left represents the initial data and sigma points. The top right represents a non-linear transition model and the bottom right represents the result of this model. The ovals represent

covariance and dots represent data points. Black represents the initial information and the UKF result. The red oval and red and blue dots represent the sigma points and the covariance respectively. The blue represents the result without the unscented transform applied.

This modification results in a new version of the Kalman filter known as the Unscented Kalman filter (UKF).

2.2.1 UKF Sigma Points

The equations for sigma point selection are shown below.

$$\lambda = \alpha * \alpha * L - L \tag{2.7}$$

$$\chi_i = \begin{cases} \bar{x}, i = 0 \\ \bar{x} + \sqrt{(L+\lambda)P}, i = 1 : L \\ \bar{x} - \sqrt{(L+\lambda)P}, i = L+1 : 2L \end{cases} \tag{2.8}$$

where L is the dimension of the state vector and α is some small value controlling the spread of sigma points. The reference state and error covariance matrix are represented by $\bar{x}$ and P respectively. Once these sigma points are propagated through the appropriate models, they are recombined with weights totaling to one.

The weights are calculated according to the following:

$$W_0^m = \frac{\lambda}{L+\lambda} \tag{2.9}$$

$$W_0^c = \frac{\lambda}{L+\lambda} + (1 - \alpha^2 + \beta) \tag{2.10}$$

$$W_i^m = W_i^c = \frac{1}{2(L+\lambda)}, i = 1 : 2L \tag{2.11}$$

where W_i^m are the first order weights and W_i^c are the second order weights. The constant β is generally set to 2 to represent a Gaussian noise.

2.2.2 UKF Predict

The modified version of the prediction stage equations 2.1 and 2.2 become:

$$\hat{\chi}_{k|k-1} = F_k[\hat{\chi}_{k-1|k-1}] + B_k u_k \tag{2.12}$$

$$\hat{x}_{k|k-1} = \sum_{i=0}^{2L} W_i^m \chi_{i,k|k-1} \tag{2.13}$$

$$P_{k|k-1} = \sum_{i=0}^{2L} W_i^c [\hat{\chi}_{k|k-1} - \hat{x}_{k|k-1}][\hat{\chi}_{k|k-1} - \hat{x}_{k|k-1}]^T + Q_k \tag{2.14}$$

where $\hat{\chi}_{k|k-1}$ represents the sigma points after being propagated through the state-transition model F. The weighted values are summed over to produce the state prediction. The prediction is subtracted from each element in $\hat{\chi}_{k|k-1}$ to find the center of the sigma points. This center is multiplied by its transpose, weighted, summed and then added to Q_k to produce the error covariance of the prediction.

2.2.3 UKF Update

The Kalman filter measurement model equation 2.3, after the transform, becomes:

$$y_{k|k-1} = H_k[\hat{\chi}_{k|k-1}] \tag{2.15}$$

$$\hat{y}_{k|k-1} = \sum_{i=0}^{2L} W_i^m y_{i,k|k-1} \tag{2.16}$$

where $y_{k|k-1}$ represents the sigma points after propagation through the state-transition model, F_k, and the measurement model, H_k. The result in the observed space is represented by $\hat{y}_{k|k-1}$. The update equations, 2.4 - 2.6, become:

$$P_{\bar{y}_k\bar{y}_k} = \sum_{i=0}^{2L} W_i^c[y_{i,k|k-1} - \hat{y}_{k|k-1}][y_{i,k|k-1} - \hat{y}_{k|k-1}]^T + R_k \tag{2.17}$$

$$P_{\bar{x}_k\bar{y}_k} = \sum_{i=0}^{2L} W_i^c[\chi_{i,k|k-1} - \hat{x}_{k|k-1}][y_{i,k|k-1} - \hat{y}_{k|k-1}]^T \tag{2.18}$$

$$K_k = P_{\bar{x}_k\bar{y}_k} P_{\bar{y}_k\bar{y}_k}^{-1} \tag{2.19}$$

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k(y_k - \hat{y}_{k|k-1}) \tag{2.20}$$

$$P_{k|k} = P_{k|k-1} - K_k P_{\bar{y}_k\bar{y}_k} K_k^T \tag{2.21}$$

where y_k represents the measurement in the observed space. Once this transformation is complete, the error covariance is no longer propagated through the models but is instead regenerated during each predict stage.

2.2.4 Cholesky Decomposition

During the generation of sigma points, the square root of the scaled error covariance matrix must be found. This square root can most easily be found by taking the Cholesky decomposition of the matrix. The algorithm only works on a Hermitian positive-definitive matrix and produces the result in a triangular matrix. The initial condition can be guaranteed in the UKF provided the noise matrices Q_k and R_k are diagonal positive definite matrices. This is because adding two Hermitian positive definite matrices produces another Hermitian positive definite matrix and multiplying any matrix by its conjugate transpose also produces a positive-definite Hermitian matrix. The solution for the Cholesky decomposition

of a four by four matrix is shown in equation 2.22

$$\begin{bmatrix} \sqrt{a_{11}} & 0 & 0 & 0 \\ \frac{a_{12}}{l_{11}} & \sqrt{a_{22} - l_{21}^2} & 0 & 0 \\ \frac{a_{13}}{l_{11}} & \frac{a_{23} - l_{21} l_{31}}{l_{22}} & \sqrt{a_{33} - l_{31}^2 - l_{32}^2} & 0 \\ \frac{a_{14}}{l_{11}} & \frac{a_{24} - l_{21} l_{41}}{l_{22}} & \frac{a_{34} - l_{31} l_{41} - l_{32} l_{42}}{l_{33}} & \sqrt{a_{44} - l_{41}^2 - l_{42}^2 - l_{43}^2} \end{bmatrix} \tag{2.22}$$

where a_{ij} represents an element in the original matrix and l_{ij} represents an element in the result.

Each element of the result in this decomposition is dependent on the results to its left and above as well as the element on the diagonal in its column. The focus will now change to information related to GPU acceleration.

2.3 GPU acceleration

This section will cover differences in CPU and GPU architectures and goals of each. It will then cover the general approach to GPU acceleration and the associated costs. The section ends with a discussion the software libraries used for acceleration.

2.3.1 CPU vs. GPU Architecture

In a CPU, a processor is generally made up of one or several cores each with a complex control block, a single arithmetic logic unit (ALU), and a unified cache or discrete caches for instructions and data. All of these cores generally share a larger unified cache. This results in an architecture where the bulk of the silicon is dedicated to controlling or supporting a relatively small section which can perform a series of computations quite quickly.

In computing, a thread is a series of instructions which are executed in order by an ALU. Threads are only aware of the data they manage and memory they access. A CPU core can normally manage several threads at once by working on one thread while others are waiting on memory accesses to complete. Programs can sometimes be accelerated on

a CPU by breaking them into multiple independent threads such that the CPU can work on one part of the program while another is delayed by memory. When program tasks do not depend on each other and can be processed in this manner it is referred to task level parallelism.

GPUs, on the other hand, are constructed to use single-instruction multiple-data parallelism to leverage cases where the same operation needs to be performed repeatedly on different sets of data. This results in an architecture where each core has a single smaller control block, a set of smaller ALUs, and a cache shared by everything in the core. This means a much larger percentage of the silicon is dedicated to computation instead of merely supporting computation.

Figure 2.2 shows the differences between the two architectures. Control blocks, ALUs, and caches are colored yellow, orange, and green respectively. The CPU has only a small portion of its space dedicated to ALUs while the GPU dedicates most of its space to computation. Though GPUs are generally clocked at a somewhat slower rate, they more than make up for it with the quantity of work done during each clock.

2.3.2 Acceleration

A section of code which must perform the same operation repeatedly on independent data likely will see a speed up from exporting the work to a GPU. This is because the GPU can operate on all of the data simultaneously in multiple threads instead of one set at a time. Though code may run faster on a GPU there are certain additional cost associated with offloading work to the GPU.

The CPU cost of sending relevant data and initiating code on the GPU may offset any performance gains from moving it to the GPU. This copy cost can sometimes be amortized by performing a batch of operations on the GPU at once. This way the CPU only incurs the copy and code setup costs a single time but can run a series of operations. This, however, may not be possible if the CPU is dependent on the GPUs result. Then the CPU code may

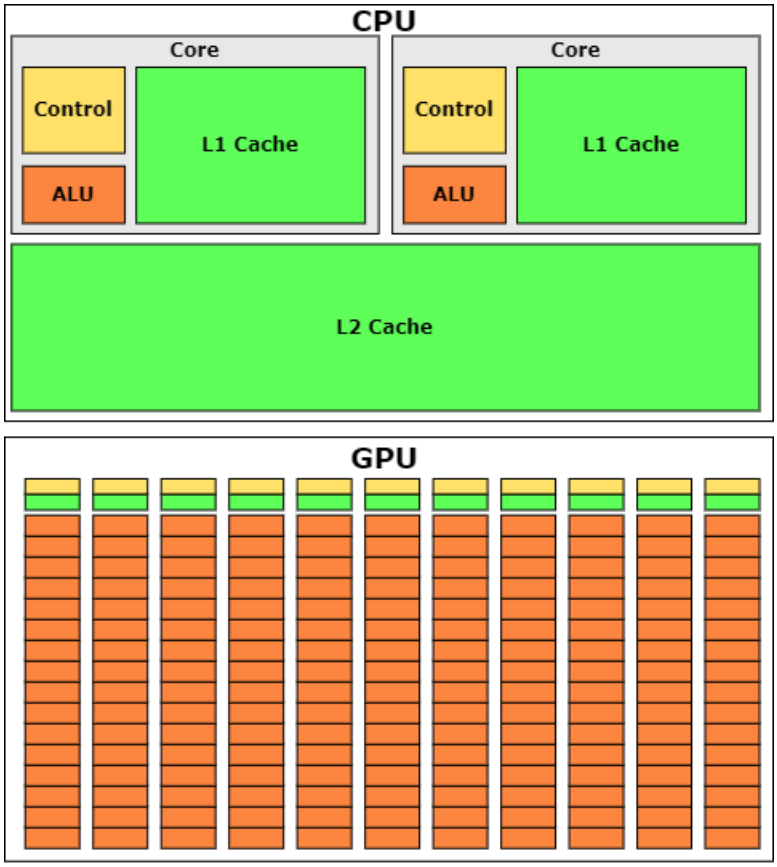

Figure 2.2: CPU vs. GPU architectures

require restructuring in order to remove the dependency and allow for concurrent CPU and GPU operation.

Code to be run on a GPU can normally be written in much the same way as code on a CPU but certain considerations must be made. In the case of a conditional where only some of the ALUs within a block meet the condition, there will necessarily be some loss in speedup over the code which is diverging. This is due to all of the ALUs being controlled by a single control unit. As illustrated in Figure 2.3, all ALUs must execute all code on each diverging branch and trash the results they do not need. Green blocks represent code blocks where the calculation occurs and results are used. Red blocks represent code which is executed but the results are not used by divergent threads in a block. If these conditionals are nested, the loss in speedup can reach the point of single threaded performance and result

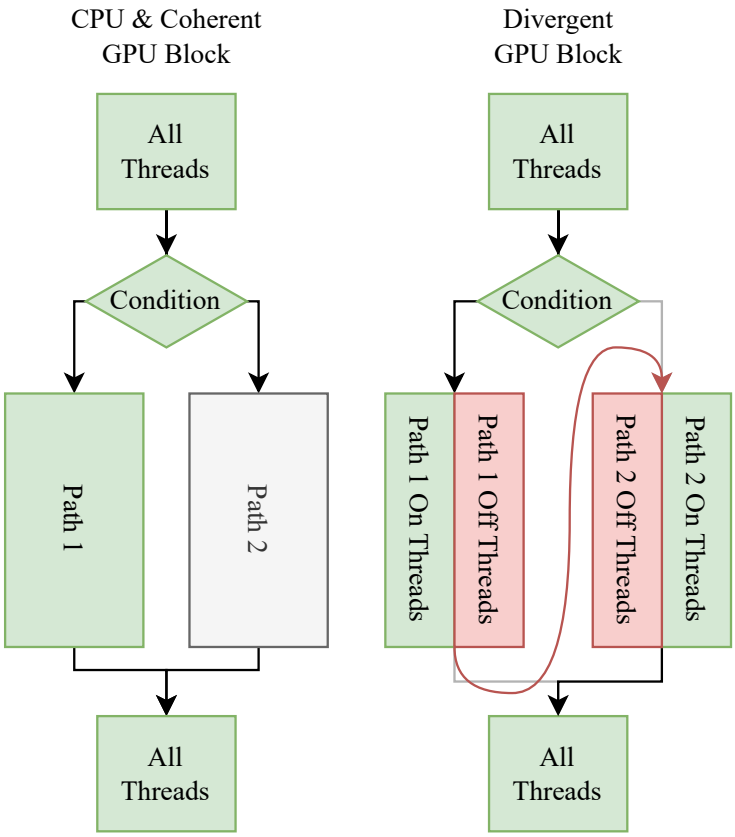

Figure 2.3: Execution flow of coherent and diverging GPU blocks.

in substantially slower execution than a CPU executing the same code. As such, care must be taken to minimize the number of diverging conditions within a single core.

Data management also changes when writing code for GPUs. Once a result is calculated in one ALU that is needed by a second ALU, the data can only be passed to the second ALU through the shared cache. This can result in delays between instructions which would not exist in a CPU.

2.3.3 CUDA

NVIDIA has developed a robust set of application program interfaces (APIs), collectively called Compute Unified Device Architecture (CUDA)[20], for general purpose GPU programming. CUDA has its own terminology for describing how a function will map to the data parallel processing units of a GPU. It refers to a function which runs on the GPU

as a Kernel, a set of ALUs managed by one control unit as a Block, and the code context on each ALU as a Thread. CUDA has a plethora of features and many are unique to its architecture. However, the features relevant to this work are standard for other GPU APIs such as OpenGL[27] and AMD's HIP[28].

The most important of these features is the *async* library. This library contains asynchronous versions of CUDA's standard library calls which do not hold control of a CPU thread while the operation completes. This means that the CPU and GPU can run concurrently instead of one at a time. While this works somewhat seamlessly for kernel calls, memory transfers require a bit more work. In CUDA, memory transfers can only occur asynchronously via direct access media (DMA) controllers if the CPU side memory is in page locked memory. Otherwise, the CPU performs the memory transfer to ensure the relevant memory is not paged out to the hard drive.

Memory in computers is generally stored in groups which go by different names in different memory levels. When the memory groups are in global memory, the name used is "page". When a page has not been accessed recently by the CPU and more space is needed, the page can be offloaded, or "paged", out to the hard drive to make room. In CUDA, this type of memory is only allocated through the cudaMallocHost() function.

Another important feature is CUDA's ability to collect work into discrete streams. This forces all work placed into a stream to happen sequentially but for work in different streams to occur in parallel. While this explicit synchronization is the only option in simpler GPU architectures, it guarantees that memory isn't transferred before it is ready and that kernels do not begin operation until its memory is ready to be used. This also allows for unrelated work to be sent to the GPU and reduce the number of idle ALUs.

2.4 Program Profiling

Valgrind[23] is a suite of tools which each analyze different aspect of a program's memory usage and thread synchronization. Of particular interest are the tools Callgrind and

Cachegrind. Together, these two tools allow for in depth analysis of code efficiency and bottlenecks on system's architecture or a system with a different cache architecture.

2.4.1 Callgrind

Callgrind[26] is a tool which assigns a cost to each function within a program. By default, the tool collects information about caller-callee function pairs and the number of assembly instructions executed by the callee. It can be told to collect a deeper call stack for specific functions, all functions, or recursive functions. It can also optionally be turned on and off to only profile costs for specific regions of code.

The output file can be digested by a companion program kcachegrind which builds a call graph of the functions and lists both the cost of each function and the cumulative cost of each function and its child functions. This allows for easy digestion of the costs and how they are distributed in code. The information collected by default is the instruction calls and data reads. It is naive about the impact of memory architecture on program efficiency. However, Callgrind is built such that it can integrate with information obtained from Cachegrind. This allows for analysis which can look at an estimate of the actual clock cycle cost of each function instead of merely the number of instructions called.

2.4.2 Cachegrind

Cachegrind[26] works by intercepting all memory accesses from a program in software. This allows for the tool to build a virtual cache to simulate how a program interacts with a different cache hierarchy. A designer can observe how a program may be causing memory issues and see opportunities for streamlining memory usage. When this program is paired with Callgrind, it allows a full stack simulation of program costs on systems with different cache architectures. This work will use this capability to simulate program performance on different boards to observe the usefulness of acceleration in different contexts.

The following chapter first describes a generic system level architecture of a PDR system. It then describes the selection representative components to implement the system level description.

Chapter 3

Pedestrian Dead Reckoning System Design

A PDR system can be broken easily into three separate components: Step Detection, Stride Length Estimation (SLE), and Heading Estimation. In a typical PDR system, step detection iterates every time new sensor information is available. Once it detects a step from the data, it triggers the SLE algorithm. The SLE algorithm analyzes the sensor data from the time range specified by step detection to produce an estimate of the distance moved in that step. Once the distance traveled is calculated, the heading values accumulated by heading detection are combined to produce a heading for that step. The heading and stride length estimate values are then combined with the previous position to produce the current position.

Each of these components has a plethora of proposed implementations, but all can be narrowed based on sensor placement. The common sensor placements include foot mounted sensors, carried, or pocketed like a cellphone, and in a chest pocket or backpack. This work targets a case that is the least intrusive to the user and provides the best performance for each of the three components of PDR. To this end, a chest mounted system was chosen. This provides a higher correlation between user walking motion and sensor detected motion than cellphone models and a higher stability for heading detection than foot mounted sensors. This positioning is also most convenient for a wearable sensor. It could be easily placed in a shirt pocket or worn like a badge.

3.1 System Architecture

The chart in Figure 3.1 shows a high level map of how the discrete PDR components are integrated into a coherent system in this work. As the sample rate of the accelerometer

and gyroscope in the test system is much faster than the sample rate of the magnetometer, the UKF only iterates when there is new magnetometer data available. The UKF output is not needed immediately so it is time stamped and stored for later.

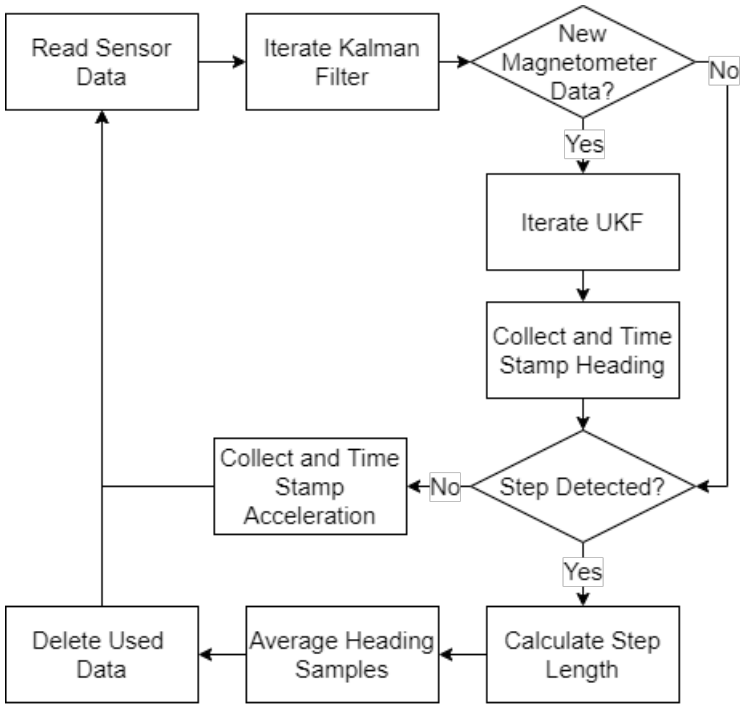

Figure 3.1: System Architecture for Integrating PDR Components

Once the system iterates the heading components appropriately, the new acceleration data is passed to the step detection component. If the step detection does not detect a step, it normally would dump the acceleration data point. However, SLE requires all acceleration data points to find the step length. So, the data is instead time stamped and collected.

If a step is detected, the system must perform a series of tasks to update its position relative to the starting point. The system first averages all of the heading values that occurred during the step and dumps all of those data points. The SLE component then calculates the length of the step and dumps all of the data it used. The heading and stride length are then combined to produce the new position. This update is always slightly behind as determined by α from equation 3.1. This is because the step detection is sure it has detected the correct peak only after it exits the prediction window from equation 3.1.

3.2 Step Detection

Solutions to PDR step detection vary from straightforward analysis of gyroscope and accelerometer signals to an in-depth analysis of the mechanics of human motion as it relates to a particular sensor placement[5][6]. Most combine some of zero cross detection, peak detection, and threshold detection of the gyroscope and acceleration signals with some more advanced signal analysis. Most of high accuracy step detection literature focuses techniques which use foot mounted sensors[18]. Those which utilized a chest mounted sensor provided no opportunity for new hardware acceleration. Additionally, some approaches were eliminated because they depend on knowledge of the user's anatomy[3]. So, selection criteria changed to ease of implementation with an adequate accuracy.

In this work we chose to implement the step detection approach proposed in [1] which uses a combination of zero-cross detection and peak detection with peak prediction on the accelerometer data to achieve an accuracy of 97.9%. Zero crossing works by finding the vector length of the accelerometer data, subtracting out a constant acceleration from gravity, and then detecting when two consecutive points straddle zero. While this is quite easy to implement, there are often false positives clumped around the true zero cross when the system is experiencing additional dynamics. For this reason, peak detection was applied to choose the zero cross closest to the midpoint of a valley and the following peak. This narrows the zero crosses to a single point on the ascending slope of the acceleration signal. Peak prediction works to eliminate false peaks by only looking for a peak in a narrow window. The window is centered on a sample $n + 1$ time steps ahead of the most recently detected peak where n is the number of samples between the previous two peaks. Algorithm operation is shown in Figure 3.2. In the case depicted, four false zeroes are eliminated with peak detection.

3.2.1 Step Detection Implementation

To operate correctly, the algorithm must know about the two previous peaks. To find these two peaks, a brute force initialization stage is used which combines peak detection and

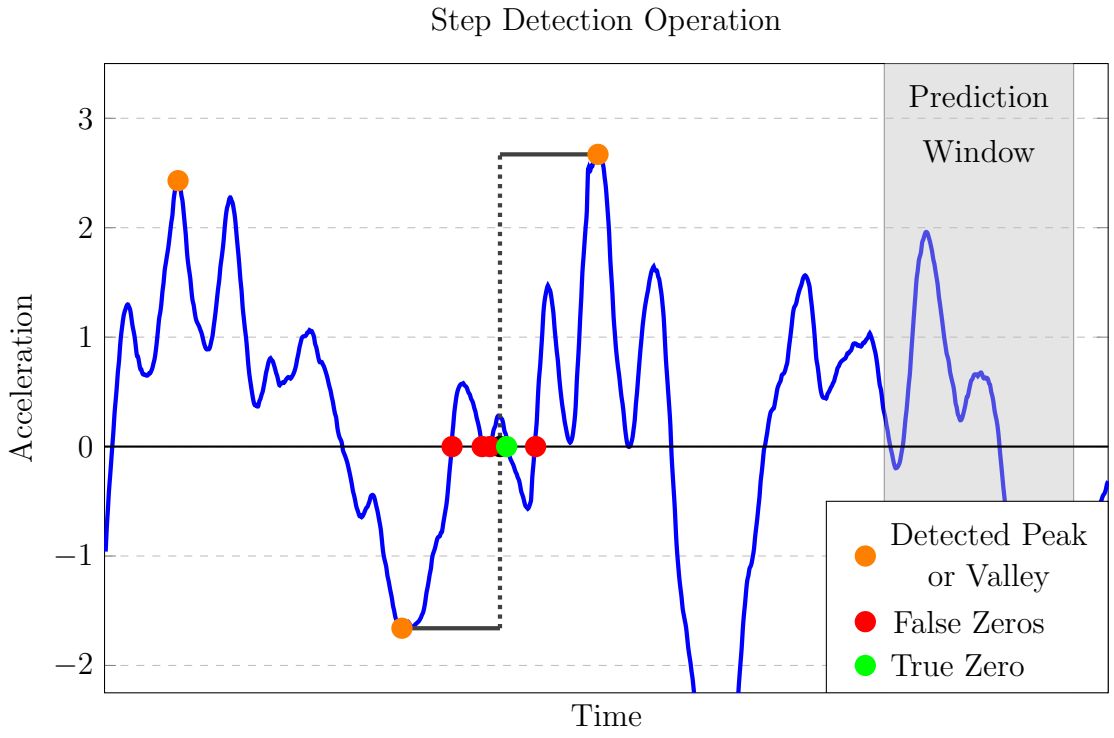

Figure 3.2: Points of interest in the step detection algorithm.

thresholding. When the acceleration rises above a threshold, the highest value seen before falling back below the threshold is considered the first peak. The same approach is used for the second peak. Until the second peak is detected, the lowest acceleration value seen is tracked and each zero cross is collected and time stamped. Once the second peak occurs, the midpoint in time between the valley and new peak is found. The zero point closest to this value is considered the start of the step. The time between the two peaks is used to find the window where the next peak should appear. Equation 3.1 is used to calculate the prediction window

$$W_{k+1} = t_k + (t_k - t_{k-1})(1 \pm \alpha) \tag{3.1}$$

where W_{k+1} represents the bounds of the prediction window and α is a value controlling the width of the window. The α chosen in this case is 0.4. This was selected as it resulted in a high reliability for step detection across the trials in our data set.

The algorithm continues in a similar fashion to the brute force approach. Except now the peaks checked are within the prediction range instead of above an arbitrary threshold. This can be seen in Figure 3.2. Additionally, if the time between peaks is longer than a chosen time, the step data is dumped, and the algorithm reverts to its initial brute force operation until a new peak is detected. This prevents abnormal data being given to SLE and producing an impossibly long step.

Due to how this algorithm is initialized, the first zero cross detected is the endpoint of the first step. This means that the data for the first step in a series can't be sent to SLE and position isn't updated. This is considered an acceptable simplification as the error measure is displacement error divided by distance traveled. So, the error from this missed step will be trivial at the end of the trial. This was verified comparing a trial with the missed step included and a trial with the missed step omitted from the reference. The root mean square (RMS) error of the position was calculated after each step. The results, shown in Figure 3.3, show that the difference in the RMS errors approaches zero over the course of the trial.

3.3 Stride Length Estimation

SLE solutions are almost exclusively based on examining attributes of the accelerometer signal over the course of a step. Common attributes examined are step frequency[5][1], the integral of the acceleration curve[6], difference between peak and valley acceleration[7][5][3][1], and acceleration sample variance during the step[3][1]. In cases where more than one of the attributes is used, the attributes are combined as a weighted linear combination[1][3]. These weights are normally trained via least mean squares(LMS) training or chosen as a function of some physical attribute, such as leg length[5][6]. Some approaches train multiple sets of weights for different modes of locomotion such as walking, running, marching, etc.[1][10] In this case, the appropriate set of weights is selected by using a neural network to classify the mode of movement.

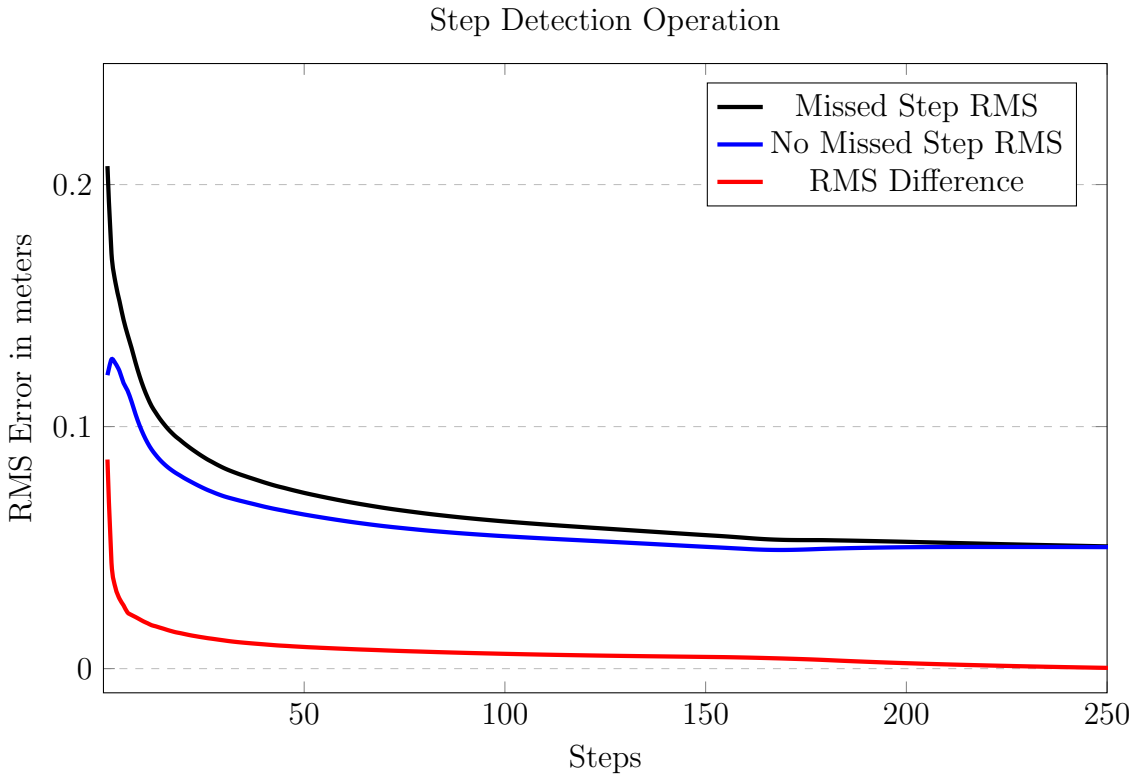

Figure 3.3: The RMS error of SLE over a long series of steps with and without a single missed step.

Though the solutions which use neural networks provide ample opportunity for hardware acceleration, neural network acceleration is well researched. As such, neural networks were ignored as a consideration when selecting a SLE solution. The remaining approaches are all fairly comparable in computation requirements so the approach for estimating stride length from walking from [1] was selected. The neural network to distinguish between the modes of locomotion was omitted. As the neural network acceleration is not of interest for this work and the data set used for testing only contains trials of walking, the SLE was reduced to only consider sensor data consistent with walking. This approach was selected due to its high accuracy and that it is from the same source from which the step detection algorithm was selected. This eased integration of data from the step detection into the selected implementation.

3.3.1 SLE Implementation

The equations describing implementation are shown in equations 3.2 through 3.5. The solution uses the difference between peak and valley acceleration, the frequency of the step, and the variance of acceleration to calculate the stride length with an error of approximately 3%. Equation 3.2 describes how the value for the different in peak and valley acceleration is calculated. Equation 3.3 describes how the frequency of a step is calculated. Equation 3.4 describes how the variance of acceleration signals during a step is calculated. Each of these step attributes is selected due to evidence that their values correlate to some degree with the length of a person's stride. Equation 3.5 describes how the values for each attribute are weighted and added to produce and estimate of a stride length.

$$d_k = \sqrt[4]{a_{max,k} - a_{min,k}} \tag{3.2}$$

$$f_k = \frac{1}{t_{k+1} - t_k} \tag{3.3}$$

$$v_k = \frac{1}{N_k} \sum_{t=t_k}^{t_{k+1}} (a_t - \bar{a_k})^2 \tag{3.4}$$

$$L = \beta d_k + \gamma f_k + \epsilon v_k \tag{3.5}$$

Where a represents an acceleration sample from the step. The starting time of step k is represented by t_k. The number of samples in a given step is N_k. The fourth root of the peak to valley difference, the step frequency, and step variance are represented by d_k, f_k, and v_k respectively. The constants β, γ, and ϵ are the values trained by the least mean square (LMS) algorithm. The LMS algorithm trains as shown below

$$\begin{bmatrix} \beta_k \\ \gamma_k \\ \epsilon_k \end{bmatrix} = \begin{bmatrix} \beta_{k-1} \\ \gamma_{k-1} \\ \epsilon_{k-1} \end{bmatrix} + j * (L_k - \hat{L}_k) * \begin{bmatrix} d_k \\ f_k \\ v_k \end{bmatrix} \tag{3.6}$$

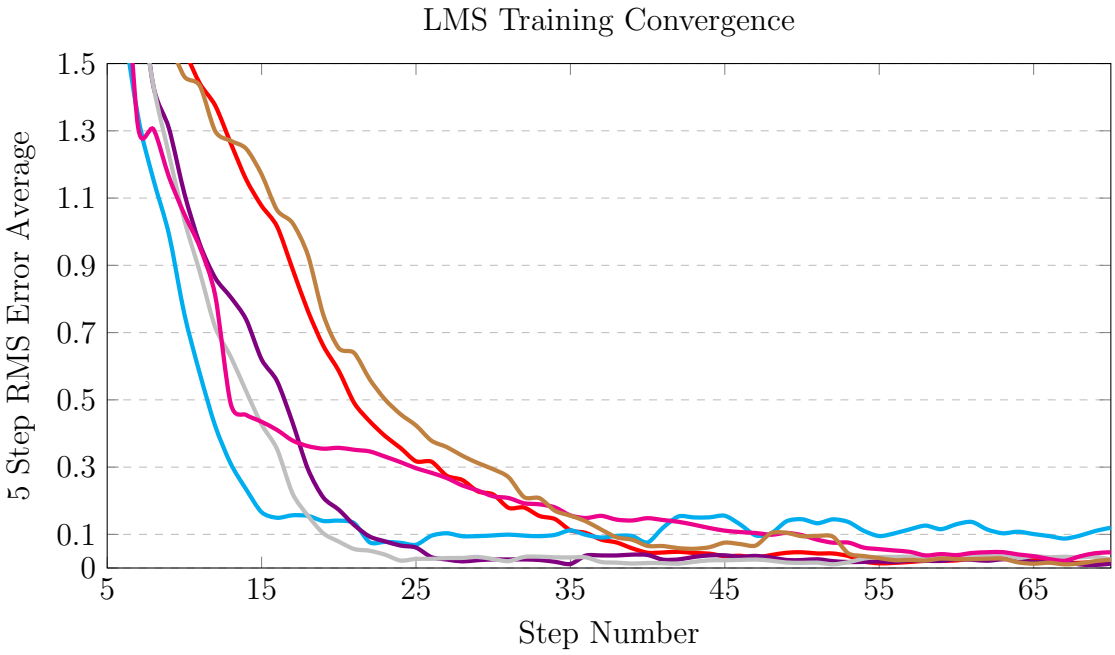

Figure 3.4: Convergence of the LMS algorithm for several trials. The plotted values are a 5 step rolling average for six different trials.

where L is the true stride length and $\hat{L}$ is the calculated stride length. The learning rate of the algorithm is represented by j. This controls how fast the algorithm converges as well as the accuracy of the final weights. Figure 3.4 shows the convergence behavior of six trials when the learning rate is set to 0.01. The results tend to reach convergence after approximately 50 steps.

3.4 Heading Estimation

Heading estimation is the process of determining the direction of travel in some global coordinate system. The approaches to heading detection can be largely broken into heading relative to magnetic north[4][2] and heading within some pre-mapped or fingerprinted region[19].

3.4.1 Fingerprinting

The fingerprinting approach to heading estimation works by first mapping a region's magnetic field or some signal network and storing it in the device. This map is then traversed using the readings from the appropriate sensor and heading is determined based on calculated travel along the map. Though this approach achieves a high degree of accuracy in research, it suffers from a couple major pitfalls.

The most obvious is that the heading algorithm is useless outside of the bounds of the mapped area. While this is acceptable in a narrow use case where the user only relies on PDR in a few particular regions, it is not a generally applicable approach.

The other pitfall is the reliance on an internal map. Though accurate shortly after creation, the topology of the map can see large changes after seemingly minor real-world disturbances. A system relying on fingerprinting must have a mechanism for regularly updating its internal map or it will become less accurate over time. This usually means a manual remapping and update of the stored map or a fixed based system which can continuously update the map. Both of these options are expensive and time consuming which would greatly hinder real-world adoption and further narrow PDR's use case.

3.4.2 Magnetic North

The other popular approach is to use readings from the accelerometer and gyroscope to orient and stabilize the magnetometer. The magnetometer provides reading which allow for the system to be globally oriented with regards to magnetic north. However, the readings from the sensor are in the device coordinate frame and subject to interference. The accelerometer readings can be used to determine the direction of gravity within the device coordinates and can therefore be used to rotate the magnetometer data into the global coordinate frame.

Though the accelerometer is stable over time, its moment to moment readings are quite noisy. This means that rotationally correcting the magnetometer with just acceleration

data introduces the same moment to moment noise into the magnetometer readings. If the gyroscope's moment to moment stability is first combined with the accelerometer's stability over time in a Kalman filter, the magnetometer is effectively shielded from the accelerometer's noise. The rotationally corrected magnetometer data can then be used as a heading.

The reading from the magnetometer can still be influenced by nearby magnetic interference. By once again introducing the gyroscope's stability in a Kalman filter, this can be alleviated. This sensor fusion then produces a magnetometer based heading which is in the global coordinate frame and robust to short term interference.

The variety of Kalman filter chosen in this case is important as the system can be quite non-linear. The UKF[4] in particular shines as it accounts for the non-linearity just as well as a particle filter with much less calculation. This system in shown in Figure 3.5.

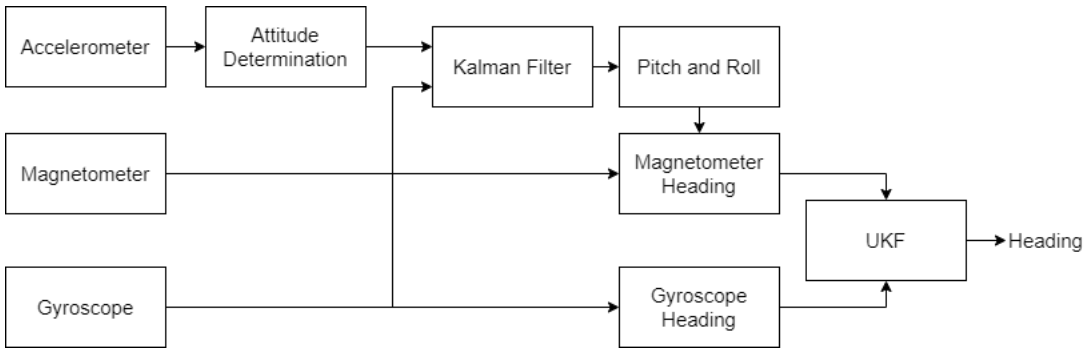

Figure 3.5: Heading implementation

This doesn't have quite the same initial accuracy of a fingerprinting approach, but it doesn't suffer from any of the glaring pitfalls and limitations either. In light of these advantages, this approach was chosen for the PDR system. The system presents opportunity for hardware acceleration which has received little attention in the literature despite the popularity of various Kalman filters for PDR in particular and embedded systems in general.

3.4.3 Heading Estimation Implementation

The first choice made in the designs of the UKF and Kalman filter was how to most easily represent three dimensional orientation. Quaternions were selected over Euler angles to represent system orientation in both filters' state. Euler angles are a set of three angles which represent orientation as quantities of rotation relative to a particular axis. However, Euler angles suffer from singularities which appear along each axis and present complications for computation. A quaternion is also a way to represent three dimensional orientation but with four components. The first component, q_w, is a scalar and the remaining three components, q_x, q_y, and q_z, represent a vector. Each vector element represents a distance along a set of three orthogonal axis in the same manner as complex numbers. Quaternions are used because they avoid the singularities of Euler angles.

However, Euler angles are much easier to understand than quaternions. For this reason, Euler angles were chosen as the output of both the Kalman filter and UKF. The Kalman filter produces only a pitch and roll angle. As a PDR user can be assumed to remain in the same general orientation while walking, Euler angles' singularities do not impact the output. In the case of the UKF, the output produced is a heading value. While this does have a singularity, the value is only used for sin() and cos() operations which are not affected by singularities.

The conversions from Euler angles to quaternions and from quaternions to Euler angles are respectively

$$\begin{bmatrix} q_w \\ q_x \\ q_y \\ q_z \end{bmatrix} = \begin{bmatrix} \cos(\phi/2)\cos(\theta/2)\cos(\psi/2) + \sin(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \sin(\phi/2)\cos(\theta/2)\cos(\psi/2) - \cos(\phi/2)\sin(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\sin(\theta/2)\cos(\psi/2) + \sin(\phi/2)\cos(\theta/2)\sin(\psi/2) \\ \cos(\phi/2)\cos(\theta/2)\sin(\psi/2) - \sin(\phi/2)\sin(\theta/2)\cos(\psi/2) \end{bmatrix} \tag{3.7}$$

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \text{atan2}(2(q_w q_x + q_y q_z, 1 - 2(q_x^2 + q_y^2)) \\ \text{asin}(2(q_w q_y - q_z q_x)) \\ \text{atan2}(2(q_w q_z + q_x q_y, 1 - 2(q_y^2 + q_z^2)) \end{bmatrix} \tag{3.8}$$

where ϕ, θ, and ψ represent pitch, roll, and yaw as Euler angles. Additionally, for a quaternion used for orientation to be considered valid, it must satisfy:

$$1 = \sqrt{q_w^2 + q_x^2 + q_y^2 + q_z^2} \tag{3.9}$$

Kalman Filter

The Kalman filter is implemented to perform sensor fusion of the acceleration and gyroscope values. This allows the stability over time of the accelerometer and the measurement to measurement stability of the gyroscope to be combined. For the predict stage, the gyroscope inputs are used to build the state-transition model F_k used in equations 2.1 and 2.2

$$F_k = \frac{1}{2} \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{3.10}$$

where ω represents the rotation measured by the gyroscope around each axis since the last iteration. The previous state $\hat{x}_{k-1|k-1}$ is also used as the input u_k. The control-input model matrix B_k is simply the identity matrix. The noise matrix Q_k is defined as a diagonal matrix of the noise of the gyroscope.

For the correct stage the acceleration Euler angles are obtained by

$$\begin{bmatrix} \text{atan2}(a_y, \sqrt{a_x^2 + a_z^2}) \\ \text{atan2}(-a_x, a_z) \\ 0 \end{bmatrix} \tag{3.11}$$

where the a_x, a_y, and a_z are the magnitudes measured by the accelerometer along each axis. These are converted to a quaternion using equation 3.7 to generate the measurement input z_k. The measurement model H_k is simply the identity matrix.

$$H_k = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.12}$$

The measurement noise matrix, R_k, is a diagonal matrix of the noise associated with the accelerometer. The new state estimate, $\hat{x}_{k|k}$ is normalized to meet the requirement set in equation 3.9. The output Euler angles can be obtained by applying equation 3.8 to the state. The Kalman filter should be initialized such that

$$\hat{x}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T \tag{3.13}$$

$$P_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

UKF

The UKF is implemented to combine the rotationally correct magnetometer heading and the gyroscope data. For the predict stage, F_k is

$$F_k = \cos(v) * I + \frac{\sin(v)}{v} * \frac{1}{2} \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{3.15}$$

$$v = \frac{\sqrt{w_x^2 + w_y^2 + w_z^2}}{2} \tag{3.16}$$

where v is half the length of the rotation vector and w_x, w_y, and w_z are the gyroscope measured rotation since the last UKF step. The control is not used so B_k can be treated as zeros. The weights W_i^m are calculated according to equations 2.9 and 2.11 with β as 2 and α as 0.1. The noise matrix Q_k is

$$Q_k = \begin{bmatrix} -q_x & -q_y & -q_z \\ q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \end{bmatrix} \begin{bmatrix} \sigma_{wx}^2 & 0 & 0 \\ 0 & \sigma_{wy}^2 & 0 \\ 0 & 0 & \sigma_{wz}^2 \end{bmatrix} \begin{bmatrix} -q_x & -q_y & -q_z \\ q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \end{bmatrix}^T \tag{3.17}$$

where the q values are taken from the *a priori* state, $\hat{x}_{k-1|k-1}$.

For the update stage, the magnetometer data can be rotationally corrected with the Euler outputs of the Kalman filter

$$\begin{bmatrix} H_x \\ H_y \\ H_z \end{bmatrix} = \begin{bmatrix} \cos\phi_{kf} & \sin\phi_{kf}\sin\theta_{kf} & -\sin\phi_{kf}\cos\theta_{kf} \\ 0 & \cos\theta_{kf} & \sin\theta_{kf} \\ \sin\phi_{kf} & -\sin\theta_{kf}\cos\phi_{kf} & \cos\phi_{kf}\cos\theta_{kf} \end{bmatrix} \begin{bmatrix} h_x \\ h_y \\ h_z \end{bmatrix} \tag{3.18}$$

where h_x, h_y, and h_z are the magnitudes measured by the magnetometer and H_x, H_y, and H_z are the rotationally corrected magnitudes. The values ϕ_{kf} and θ_{kf} are obtained from the Kalman filter. The magnetometer heading input, y_k, is then obtained by:

$$y_k = \begin{bmatrix} 0 \\ 0 \\ \text{atan2}(H_y, H_x) \end{bmatrix} \tag{3.19}$$

The measurement model, H_k, is simply equation 3.8. The process to obtain the measurement noise, R_k, is

$$M_{\phi\theta} = \begin{bmatrix} 0 & \frac{a_z}{a_y^2+a_z^2} & -\frac{a_y}{a_y^2+a_z^2} \\ -\frac{\sqrt{a_y^2+a_z^2}}{a_x^2+a_y^2+a_z^2} & \frac{a_x a_y}{(a_x^2+a_y^2+a_z^2)\sqrt{a_y^2+a_z^2}} & \frac{a_x a_z}{(a_x^2+a_y^2+a_z^2)\sqrt{a_y^2+a_z^2}} \end{bmatrix} \tag{3.20}$$

where a_x, a_y, and a_z are the accelerometer readings and $M_{\phi\theta}$ is an intermediate value used to clarify equations. The covariance of pitch and roll can be found as

$$\sigma_{\phi\theta}^2 = M_{\phi\theta} \begin{bmatrix} \sigma_{a_x}^2 & 0 & 0 \\ 0 & \sigma_{a_y}^2 & 0 \\ 0 & 0 & \sigma_{a_z}^2 \end{bmatrix} M_{\phi\theta}^T \tag{3.21}$$

where σ_{a_x}, σ_{a_y}, and σ_{a_z} are the variance of each accelerometer axis. The variance of the magnetometer based heading can be found as

$$M_\psi = \begin{bmatrix} -\frac{H_y}{H_x^2+H_y^2} & \frac{H_x}{H_x^2+H_y^2} \end{bmatrix} \begin{bmatrix} \cos\theta_{kf} & \sin\theta_{kf}\sin\phi_{kf} & \cos\phi_{kf}\sin\theta_{kf} \\ 0 & \cos\theta_{kf} & \sin\theta_{kf} \end{bmatrix} \tag{3.22}$$

$$\sigma_\psi^2 = M_\psi \begin{bmatrix} \sigma_{h_x}^2 & 0 & 0 \\ 0 & \sigma_{h_y}^2 & 0 \\ 0 & 0 & \sigma_{h_z}^2 \end{bmatrix} M_\psi^T \tag{3.23}$$

where H_x and H_y are from equation 3.18 and θ_{kf} and ϕ_{kf} are from the Kalman filter Euler output. The value M_ψ is an intermediate value used to clarify equations. The variance of noise along each magnetometer output are represented as $\sigma_{h_x}^2$, $\sigma_{h_y}^2$, and $\sigma_{h_z}^2$. Measurement noise matrix, R_k, can finally be generated

$$R_k = \begin{bmatrix} \sigma_{\phi\theta}^2[1][2] & 0 & 0 \\ 0 & \sigma_\psi^2 & 0 \\ 0 & 0 & \sigma_C^2 \end{bmatrix} \tag{3.24}$$

where σ_C^2 is set to 10^{-5}. The variable $\sigma_{\phi\theta}^2[1][2]$ represents either of the opposite diagonal elements of $\sigma_{\phi\theta}^2$. The filter is initialized to

$$\hat{x}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T \tag{3.25}$$

$$P_0 = \begin{bmatrix} 10^{-5} & 0 & 0 & 0 \\ 0 & 10^{-5} & 0 & 0 \\ 0 & 0 & 10^{-5} & 0 \\ 0 & 0 & 0 & 10^{-5} \end{bmatrix} \tag{3.26}$$

The following chapter discusses the collection of initial profiling information collected and its analysis. It then describes the architecture targeted during acceleration. It then describes the low level steps taken to accelerate the chosen PDR component. It concludes with the system level restructuring done to remove data dependencies between the CPU and GPU.

Chapter 4

Acceleration of the Unscented Kalman Filter

This chapter first covers the results of profiling the software implementation of PDR with Valgrind tools. The architecture targeted by the Valgrind simulations and GPU code design is discussed. Its memory architecture determines the scale of the overheads incurred by the CPU when starting a job on the GPU. The memory architecture of the GPU and arrangement of its cores impact the degree of parallelism achievable and the way data is handled within the UKF kernel.

4.1 Profiling Results

The first step in determining what portions of a program to accelerate is to analyze it with a software profiler. To this end, the PDR system code was run through Callgrind with Cachegrind enabled. For the purpose of hardware acceleration, the inclusive cost of a function is the metric of interest. This is the number of CPU clock cycles spent inside of a function and each of its child functions.

The top results by inclusive cost are shown in Table 4.1. The values for step detection and SLE are also included to show their relative impact on performance. Immediately clear from table is that heading estimation consumes 83.75% of the processor's clock cycles. Most of this is spent within the UKF. Additionally, most UKF cycles are spent in the state transition and measurement functions which implement equations 3.15 and 3.8 respectively.

With these results in mind, the UKF is the clear choice to investigate for acceleration. The bulk of the operations performed in the UKF are matrix multiplications. These accelerate well as each element of the result can be calculated independently of each other with

Function Name	% of Total Cost	# of Executions	Component
UKF	48.43	63,909	Heading
Kalman filter	35.32	261,999	Heading
UKF::Predict	23.26	63,909	Heading
UKF::Correct	18.97	63,909	Heading
UKF::StateTransitionFunction	16.45	575,181	Heading
UKF::MeasurementFunction	9.28	575,181	Heading
ReadSensors	9.06	1,763,721	System
UKF::CalculateR	2.92	63,909	Heading
UKF::SigmaPoints	2.47	127,818	Heading
Step Detection	0.69	261,999	Step Detection
SLE	0.6	997	SLE

Table 4.1: The top Callgrind results sorted by their inclusive cost. SLE and step detection costs are included for reference.

an identical series of operations. This means that, each of these elements can be calculated in parallel on the GPU by leveraging their data parallelism. The state transition function will map nicely to the GPU as it operates independently on each sigma point. Once the F_k matrix is constructed, each element of each sigma point can also be calculated independently. The measurement function operates on each sigma point independently like the state transition function, but the process to calculate each element within a sigma point is substantially different. So, data parallelism cannot be exploited within a particular sigma point but can between discrete sigma points. If handled carefully, the matrix inversion from equation 2.19 can be calculated completely in parallel.

Overall, the UKF maps quite well to a GPU except for the calculation for R_k. This calculation appears to contain many matrix operations but the reduced form of each of its elements shows the minimum required operations exhibit no data level parallelism.

In order to run correctly with Callgrind, the code was compiled with the flags *-g -O1*. The flag *-g* adds in code for debugging. This slows down the code for normal runs, but also allows Callgrind to determines what instructions belong to which functions. The *-O1* flag is used to perform a small amount of optimization. Generally, this optimization is not required

to use Callgrind effectively. Due to some unknown quirk with the CPU architecture, the results were unusable if this was not included.

4.2 Target Architecture

To take an informed approach to our selection of hardware acceleration strategies, we first examine our target architecture. For the purpose of development, NVIDIA's Jetson Nano was selected. The Jetson uses NVIDIA's X1 system on a chip (SoC). This means the CPU and GPU are on the same piece of silicon which allows for faster communication than a typical architecture. The Jetson's CPU is made up of four ARM Cortex -A57 cores, each with a 48KB L1 instruction cache and 32KB L1 data cache, and a unified 2MB L2 cache. Its GPU is an implementation of NVIDIA's Maxwell architecture with four regions of 32 cores. A diagram of the full SoC can be seen in Figure 4.1. The CPU complex is shown in purple while the GPU is shown in green.

The internal layout of the Maxwell GPU is shown in Figure 4.2. The light green blocks are compute cores which handle up to single precision or 32-bit floating point operations. The dark green blocks are load store units which handle interaction with the memory hierarchy. The green blocks are special function units (SFU) which perform operations like sin() and cos() on 32-bit floating point values. The light blue blocks are data caches, instruction caches, instruction buffers, and 96KB shared memory in order of first appearance from top to bottom. The dark grey represents the 32-bit registers associated with a block. The orange and dark orange blocks handle instruction scheduling and dispatching. These are equivalent to the control units from Figure 2.2. The yellow block is the Polymorph Engine which is designed specifically for graphics rendering tasks and is irrelevant to this work. The dark blue blocks are texture units which are designed to wrap textures onto 3D meshes and are also irrelevant for this work.

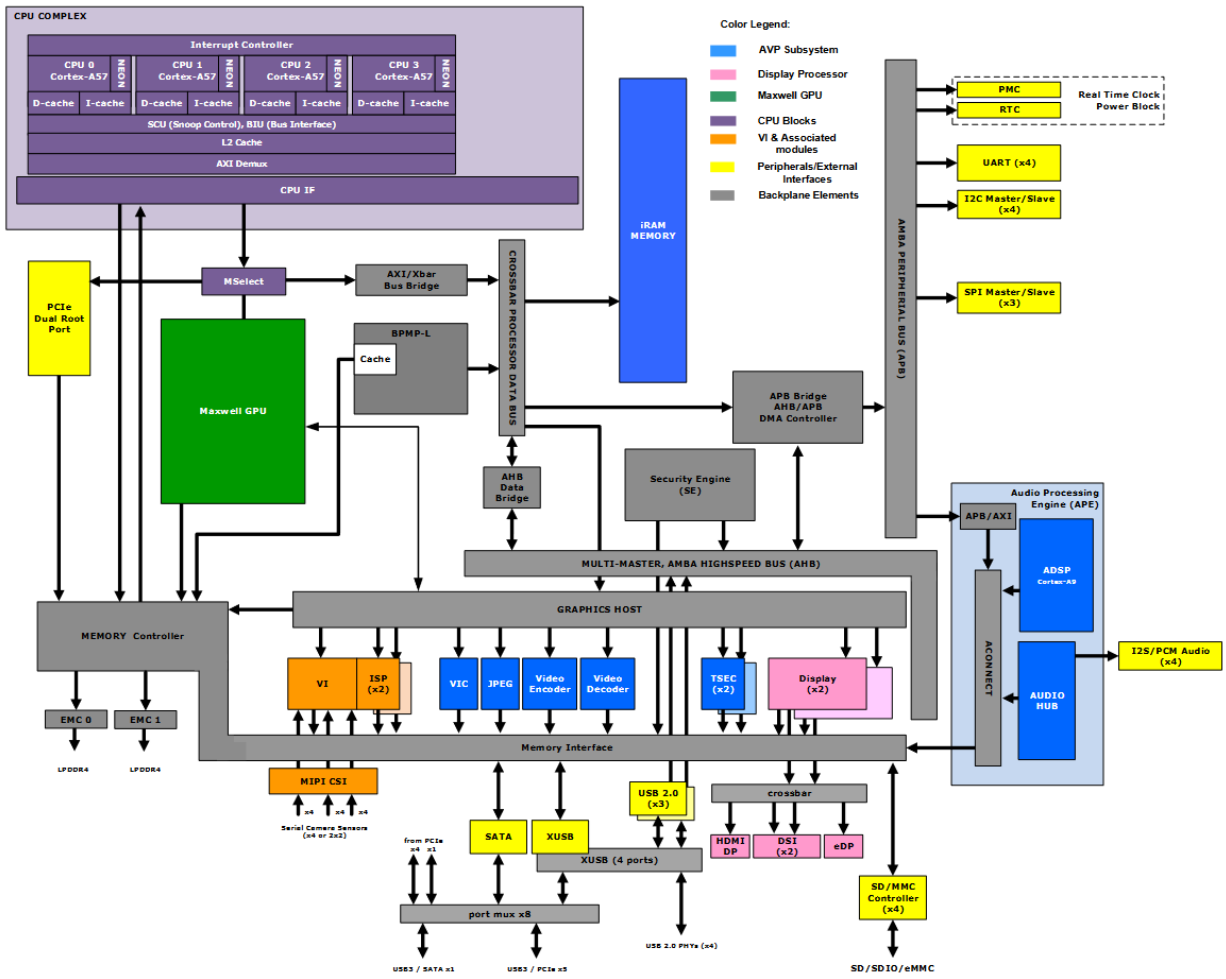

Figure 4.1: NVIDIA's Jetson Nano Architecture[22]

Each of the four regions in the GPU can manage up to 8 computational blocks at a time. So, the SoC's GPU can support up to 32 blocks of up to 32 parallel threads at once. For the purpose of this work, the 32 threads per block is the more important limit.

4.2.1 GPU Memory Architecture

Several types of data memory exist within the X1 SoC GPU's memory model. From highest to lowest these are the register file, shared memory, L1 Cache, and global memory where higher means a shorter access time. This is described in Figure 4.3.

The register file is the working space for the cores. Each thread is assigned some block within the register file where it can store its local variables. Values within the register file

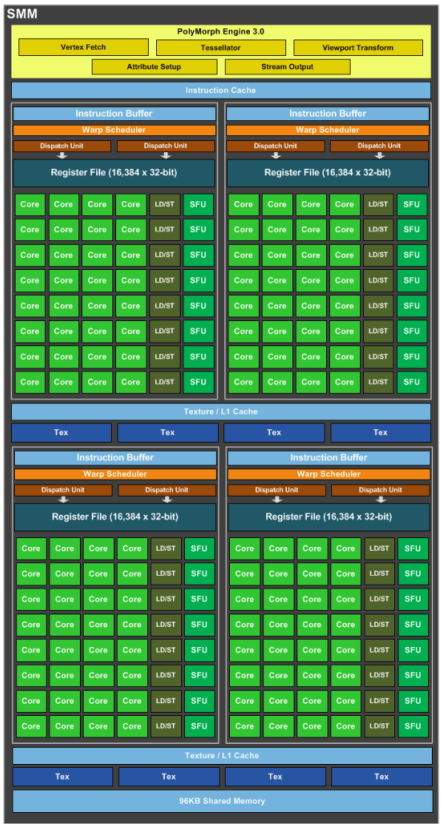

Figure 4.2: NVIDIA's Maxwell Architecture[21]

cannot be shared with other threads without first being transferred to a lower memory level. So, it is advantageous to design threads such that they maximize the work done before needing to share memory with other threads.

The shared memory is the working space for a block of threads. Shared memory is designed to allow for parallel access by threads in a block without generating conflicts. This is the fastest memory accessible across threads but data cannot be shared across blocks of threads without first being transferred to a lower memory. Kernels should be designed to complete as much work in shared memory as possible before sharing across blocks.

The L1 cache is equally as fast as shared memory but is restricted to storing data which is declared as constant. This is not useful to a system like PDR without substantial architectural changes to the design described in Figure 3.1.

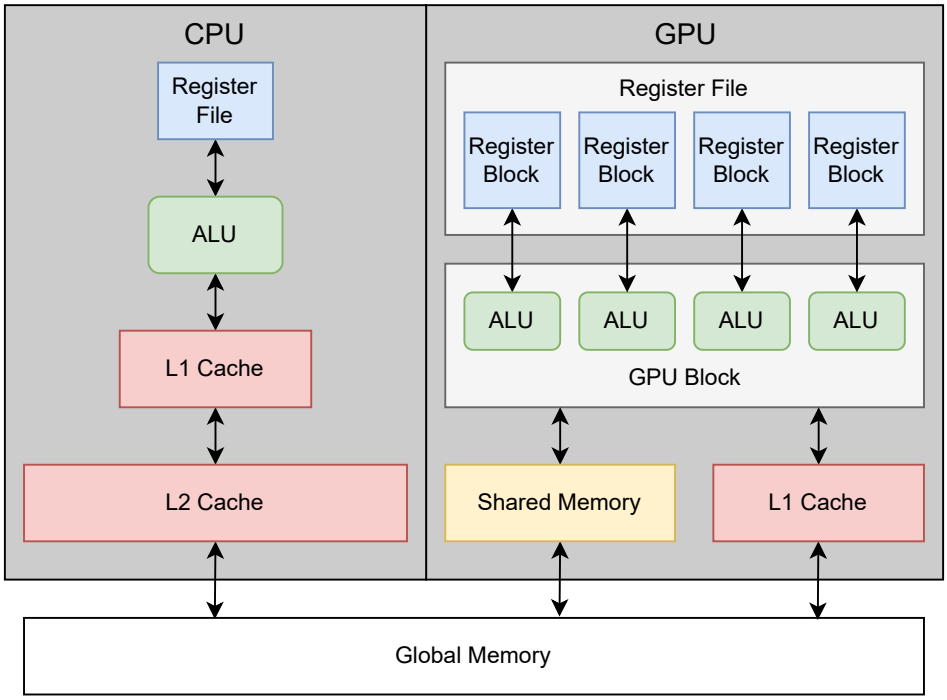

Figure 4.3: The memory organization of NVIDIA's Jetson Nano.

Global memory is the highest level of memory which is accessible by both the GPU and CPU. Regions within this level are assigned to be accessible by the GPU or CPU but not both. For data to be transferred between the CPU and GPU, it must be copied from a region owned by one into a region owned by the other. This copy, combined with the cost of sending memory up and down the two memory hierarchies, is the source of a large overhead for GPU kernels.

By default, CUDA forces the CPU to handle data transfers between discrete regions of global memory. This prevents the memory manager from sending, or "paging", the CPU data out to the hard drive during a copy operation. However, this also prevents the CPU from doing any useful work while the memory is being copied and further exaggerates the overhead of copying memory between processors. This explains the importance of transferring memory with page locked CPU regions which are allocated with cudaMallocHost(). The CPU can then continue doing work while the DMA controllers handle the memory copy.

4.3 Software Restructuring

If the program were left structured the same as described in Figure 3.1, which is repeated below, the CPU would still be required to wait for data to be copied to the GPU, the GPU to evaluate the UKF, and the resulting output to be copied back to the CPU. In order to remove this dependence, the software architecture was changed at a system level. Figure 4.4 shows the system level results of the restructuring. The chart is segregated into CPU and GPU tasks with asynchronous DMA memory transfers bridging the gap.

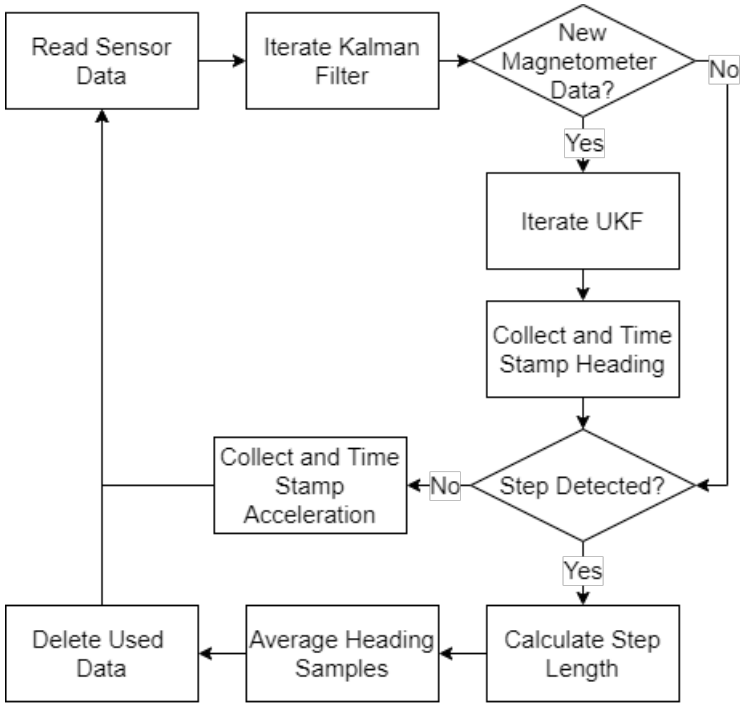

Figure 3.1: System Architecture for Integrating PDR Components

The Iterate UKF block described in Figure 4.4 can be seen in Figure 4.5. This shows the divide between the calculation for R_k and the rest of the UKF. The numbers in parenthesis correspond to the equations which are evaluated in the block. The left side is iterated over in the CPU until a step is detected. The data is then copied to the global memory for the GPU to use. The GPU then iterates over the right side until no data remains.

To accomplish this, the rotationally corrected magnetometer heading from 3.19, the gyroscope readings, and the diagonal of the measurement noise matrix, R_k, were stored in

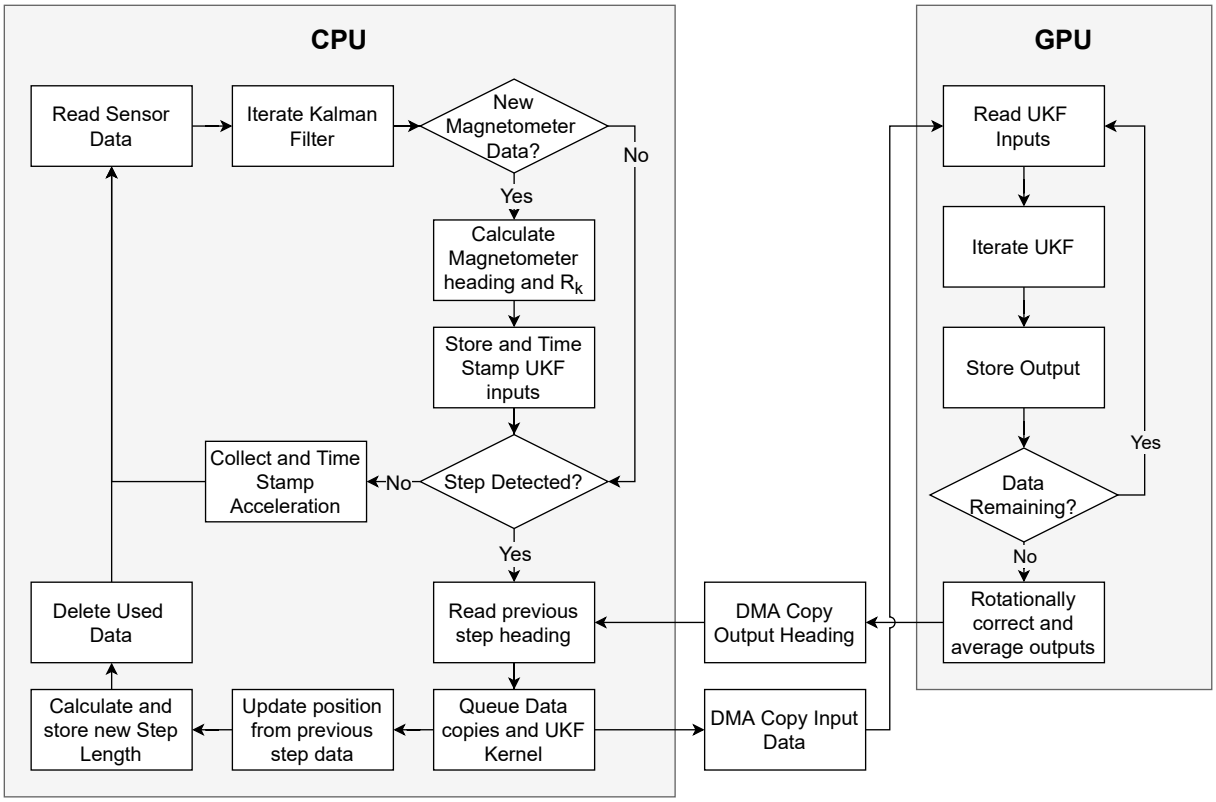

Figure 4.4: The PDR system after restructuring to decouple the CPU and GPU

CPU page locked memory when the UKF would originally have been iterated. When a step is detected, all the values are copied asynchronously to the GPU and the ones corresponding to the step are deleted from CPU memory. A modified version of the UKF kernel is queued followed by a copy of the step's heading into CPU memory. After these are queued, the CPU continues while the GPU concurrently iterates the UKF. The output of the SLE algorithm is stored instead being used as the heading for the step is still being calculated in the GPU. The next time a step is detected, the heading from the previous step is waiting in CPU memory due to the asynchronous memory copy. This value is then combined with the previous SLE output to update the position. This restructuring means there is an additional single step delay in the update in position.

The UKF kernel was modified to iterate over all of the data instead of terminating after a single iteration. The output from each iteration is stored in global memory instead of shared memory because CUDA does not allow allocation of shared memory of a dynamic

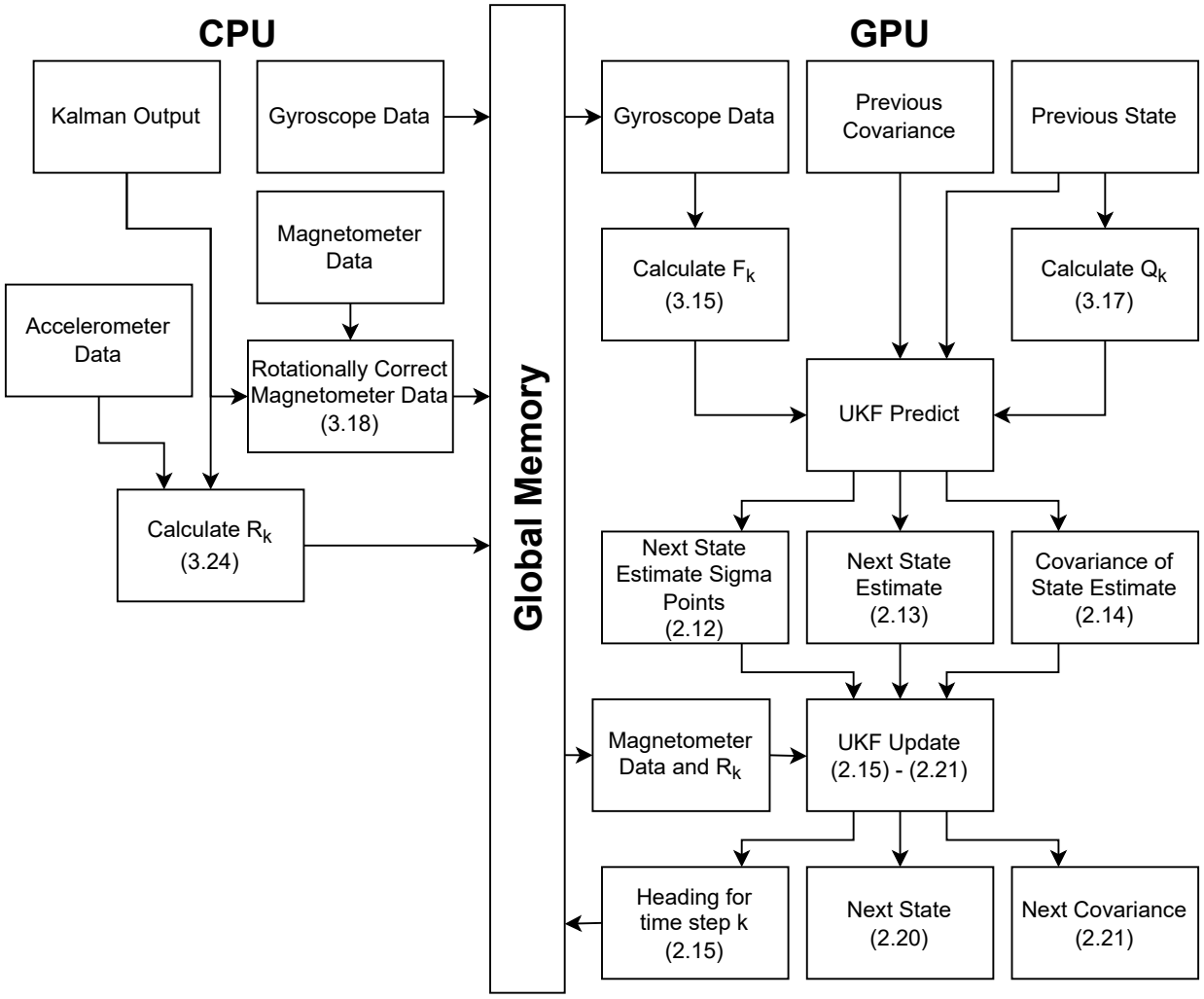

Figure 4.5: The UKF Iterate block broken between CPU and GPU.

size. After all iterations are complete, the outputs are rotationally corrected in parallel to match the orientation of the first UKF iteration. They are then summed and averaged in a binary fashion similar to the process used to average sigma points. Once this averaging is complete, the resulting state is stored in global memory to be copied to the CPU. This modification has the additional benefit of allowing intermediate state and error covariance data to be kept in shared memory instead of global memory after each iteration.

The next chapter will begin by describing how performance gain from the GPU acceleration was collected. The results will then be listed and discussed. The process of simulating the original CPU code for other embedded processing systems will be described and results

analyzed. An argument will be made for the usefulness of this work for the simulated systems in light of differences in GPU design.

4.4 Acceleration

Inspection of the generic equations for a UKF, equations 2.12 through 2.21, and the chosen implementation, equations 3.15 through 3.24, shows that most operations are matrix multiplication and addition. These types of operations exhibit a high degree of data level parallelism, so they tend to map nicely to a GPU. Additionally, the state transition and measurement models are both evaluated nine times over a different sigma point each time they appear. This presents an added layer of exploitable data level parallelism.

4.4.1 Thread Geometry

In order to decide on the quantity and arrangement of threads, the maximum amount of data parallelism in the UKF was found. The propagation of sigma points through the state transition function is the step where this occurs. This operation exhibits 36 way parallelism as the calculation of each element of each sigma point is data independent from all other elements. The choice was made to exploit only half of this parallelism in order to keep all calculation within a single 32 thread block. This way costs from data synchronization can be restricted to the shared memory. We elect to split the parallelism through the sigma points. In other words, the first two elements of each sigma point are calculated first, and the second two elements are calculated second. This results in an arrangement of two rows of nine threads.

This arrangement presents a problem when producing a matrix which has fewer than 18 elements but is larger than two in each dimension. If the matrix has a dimension of size one or two, then the thread row number can be used as that dimension's index and the thread column number can be used as the index for the other dimension. If both dimensions are larger than two, the use of thread indices becomes more complex. Every case in which this

occurs produces a matrix with, at most, length of four in its longest dimension. To deal with this case, the threads were mapped into the elements of a four by four matrix using Algorithm 1 where i_{2x9} and j_{2x9} represent the original thread row and column indices. The

Algorithm 1 Thread Mapping

$i_{4x4} \leftarrow i_{2x9} * 2$
$j_{4x4} \leftarrow j_{2x9} \pmod 4$
if $j_{2x9} \geq 4$ **then**
    $i_{4x4} \leftarrow i_{4x4} + 1$

variables i_{4x4} and j_{4x4} represent the thread row and column indices in their transformed arrangement. This operation is shown visually in Figure 4.6. Now, i_{4x4} and j_{4x4} can be used

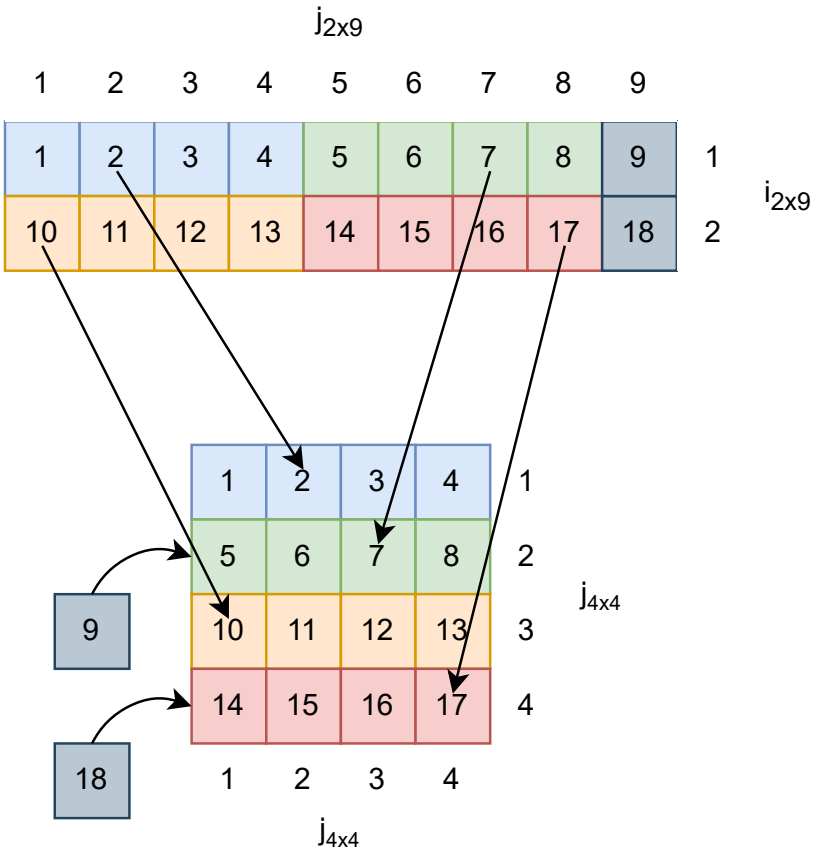

Figure 4.6: Mapping of a block of 2x9 threads to a block of 4x4 threads.

within the threads to simulate the indices of a four by four thread arrangement. Care must

Measurement Period	CPU Only	GPU Acceleration	% Slower
Average Per Step (ms)	0.29301	1.5901	442.68%
Total Time (s)	18.73	101.62	442.55%

Table 4.2: PDR Execution Time for GPU port with discrete kernel call per equation.

be taken to when using these as the fifth column and last column of threads map to the same set of indices.

The degree of leveraged parallelism in the UKF varies from step to step. Normally, this is largely handled by creating a separate kernel each time this occurs. For the UKF, this would mean a separate kernel for each equation queued sequentially by the CPU with memory transfers at the beginning and end of the series. Each of these queues in the CUDA API comes with a CPU overhead that together take more time than the CPU performing the UKF calculations itself. This is shown in Table 4.2 where the first column represents the execution time of PDR without the GPU code and the second is of GPU code with discrete calls. The third column shows that the cost of several kernel calls per UKF iterations make this approach impossible.

In order to deal with this problem, IF statements were used to effectively disable unneeded threads within one kernel. This traded the overhead from each CPU kernel call per UKF iteration for the overhead of executing an IF statement on the GPU. The one remaining kernel call for each UKF iteration was further removed by the system reorganization discussed in the previous section.

While conditionals can cause execution path divergence when not all threads take the same execution path, a single IF statement can be considered a special case. When divergence occurs, the different execution paths are each iterated over sequentially. Since a single IF statement normally simply skips ahead when the condition is not met, there is no other divergent path to concatenate with the path taken by the other threads. Figure 4.7 shows Figure 2.3 modified to include this special case of thread divergence. Some threads execute along the first path ignoring their results like before, but there is no additional execution path

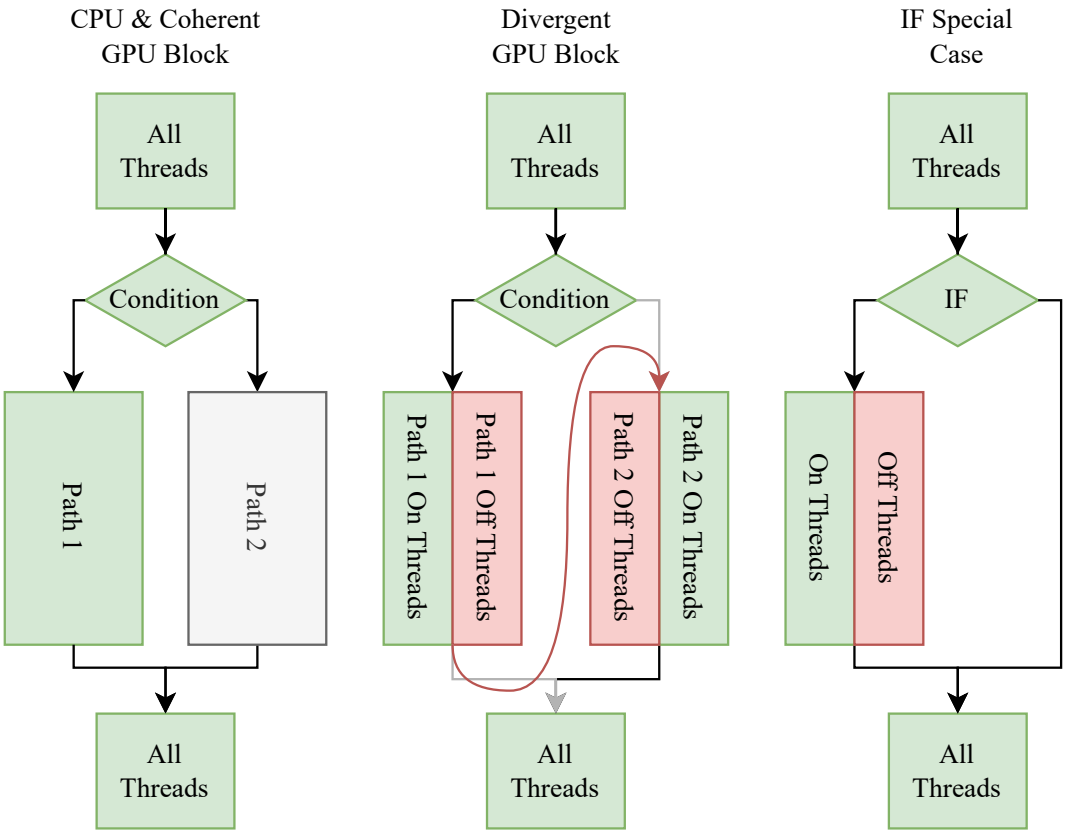

Figure 4.7: A special case of a divergent GPU block which can be utilized to effectively deactivate threads.

that must also be traversed. This allows for functionally changing the number of threads depending on how much parallelism is present.

4.4.2 Matrix Multiplication

Due to the quantity of matrix operations and the computational similarity of each, it was advantageous to take a generalized approach to their implementation. Though certain special cases exist which can be leveraged to further optimize computation, this approach applies to most cases. As illustrated in Figure 4.8, each element in the result of a matrix multiplication is independent and only needs a single row or column from each operand matrix. So, one thread is assigned to calculate each result element of the matrix and remaining threads are deactivated. Shared memory's parallel data loading can be leveraged to access operands simultaneously as well as write the final results simultaneously.

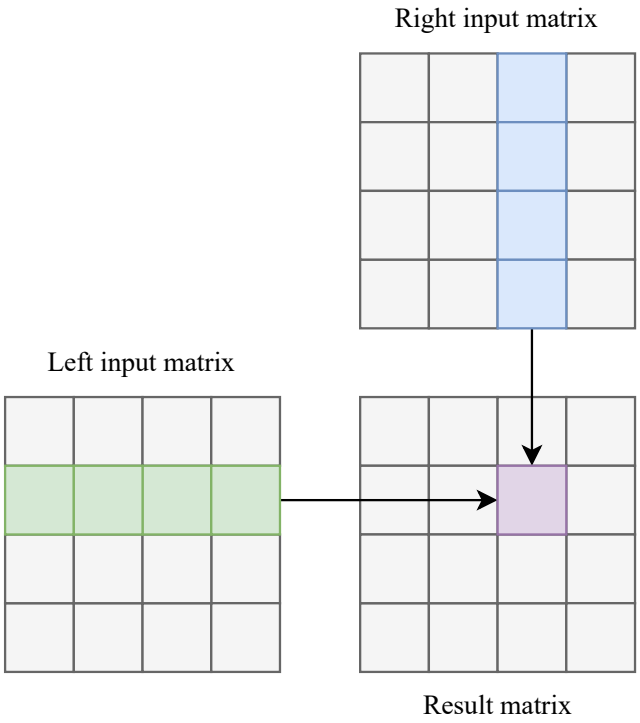

Figure 4.8: A description of matrix multiplication highlighting its parallelism.

A technique commonly used for loop acceleration in CPUs called loop unrolling is also used here and throughout the UKF acceleration. When the number of iterations of a loop is known, the cost of checking the loop condition and jumping to the beginning of the loop can be removed by calling the loop body repeatedly. This works well for our case as the number of multiplications performed for each result element is known. This can be seen in Algorithm 3 where each multiplication is performed separately instead of in a loop.

A library exists within CUDA called cuSolver for generalized matrix operations on the GPU for all NVIDIA GPU architectures. The library also performs tasks such as data validation and work segmentation which are not required for this use case. For these reasons, it was decided that this library should not be used. Each of the features listed constitutes a substantial overhead for this implementation. An implementation of the matrix operations in a single kernel also allows for all matrix results to be kept in shared memory instead of global memory.

4.4.3 Sigma Point Creation

The first step in UKF is to calculate the sigma points from the state and error covariance according to equation 2.8, which is repeated here for reference.

$$\chi_i = \begin{cases} \bar{x}, i = 0 \\ \bar{x} + \sqrt{(L+\lambda)P}, i = 1 : L \\ \bar{x} - \sqrt{(L+\lambda)P}, i = L+1 : 2L \end{cases} \tag{2.8}$$

To do this, the Cholesky decomposition of the scaled error covariance must be calculated. As mentioned in section 2.2.4, each element in the result depends on each result to its left and above as well as the diagonal value in its column. So, the decomposition can be found by having each thread in a column calculate the diagonal value. The thread corresponding to the diagonal is deactivated and the columns calculate their respective values. The whole column then stores its result. The algorithm proceeds from left to right to solve for each consecutive column. Every thread in a column calculates the diagonal value to avoid needing to pass data through the shared memory. Figure 4.9 shows which elements of the result are solved for in each step of the algorithm. This illustrates the peak degree of parallelism seen is three.

1			
2	3		
2	4	5	
2	4	6	7

Figure 4.9: The results solved for in each step of the Cholesky decomposition

The four by four grid of threads then adds or subtracts the results in parallel according to equation 2.8 to produce a 4x9 matrix of sigma points, $\hat{\chi}$, in shared memory.

The quaternions produced for each sigma point after propagation through the state transition model do not satisfy the constraint stated in equation 3.9. However, the statistical properties of the sigma points would be distorted or destroyed through normalization. So, to preserve these properties, quaternion normalization is only applied to the sigma points before the conversion to Euler angles in the measurement model and to the *a posteriori* state, $\hat{x}_{k|k}$.

4.4.4 State Transition Iteration and State Estimation

In order to propagate the sigma points through the state transition function in equation 2.12, the F_k matrix must be first constructed in shared memory according to equation 3.15. These are repeated below for reference.

$$\hat{\chi}_{k|k-1} = F_k[\hat{\chi}_{k-1|k-1}] + B_k u_k \tag{2.12}$$

$$F_k = \cos(v) * I + \frac{\sin(v)}{v} * \frac{1}{2} \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{3.15}$$

In order to accomplish this in parallel, the arrangement of the gyroscope data sent to the GPU was changed to allow for convenient parallel access. The new ordering, $[0, w_x, w_y, w_z, w_y, w_x]$ enables each thread to access the correct value for the F_k matrix by simply summing their $4x4$ indices. The value v is calculated separately in each thread according to equation 3.16. The full algorithm for generating matrix F_k is described in Algorithm 2. The value d represents the input gyroscope data for each axis where w_x, w_y, and w_z are the gyroscopes measurements around each respected axis. The variable v represents the length of the rotation vector. The variable F represents the state transition model, F_k being constructed.

Algorithm 2 Constructing F_k

Require: $d = [0, w_x, w_y, w_z, w_y, w_x]$
$v = \sqrt{d[1] \times d[1] + d[2] \times d[2] + d[3] \times d[3]}$
$F[i_{4x4}][j_{4x4}] \leftarrow d[i_{4x4} + j_{4x4}] \times \sin(v) \div v \div 2$
if $(j_{4x4} = i_{4x4} - 1)$ & $j_{4x4} \neq 0$ **then**
$F[i_{4x4}][j_{4x4}] \leftarrow -F[i_{4x4}][j_{4x4}]$
if $i_{4x4} = j_{4x4}$ **then**
$F[i_{4x4}][j_{4x4}] = \cos(v)$

Now the sigma points can be state transitioned by simply multiplying the F_k matrix by the $\hat{\chi}_{k-1|k-1}$ matrix of sigma points. As described previously, the first two elements of each point are calculated in parallel followed by the second two elements. This process is shown in Algorithm 3 with the loops unrolled. The F array represents the state transition model just constructed. The arrays $\hat{\chi}_{k-1|k-1}$ and $\hat{\chi}_{k|k-1}$ represent the matrix of sigma points before and have propagation though the model. The variable *acc* simply accumulates the intermediate result.

Algorithm 3 State Transition

$acc = F[i_{2x9}][0] \times \hat{\chi}_{k-1|k-1}[0][j_{2x9}]$
$acc = acc + F[i_{2x9}][1] \times \hat{\chi}_{k-1|k-1}[1][j_{2x9}]$
$acc = acc + F[i_{2x9}][2] \times \hat{\chi}_{k-1|k-1}[2][j_{2x9}]$
$\hat{\chi}_{k|k-1}[i_{2x9}][j_{2x9}] = acc + F[i_{2x9}][3] \times \hat{\chi}_{k-1|k-1}[3][j_{2x9}]$
$acc = F[i_{2x9} + 2][0] \times \hat{\chi}_{k-1|k-1}[0][j_{2x9}]$
$acc = acc + F[i_{2x9} + 2][1] \times \hat{\chi}_{k-1|k-1}[1][j_{2x9}]$
$acc = acc + F[i_{2x9} + 2][2] \times \hat{\chi}_{k-1|k-1}[2][j_{2x9}]$
$\hat{\chi}_{k|k-1}[i_{2x9} + 2][j_{2x9}] = acc + F[i_{2x9} + 2][3] \times \hat{\chi}_{k-1|k-1}[3][j_{2x9}]$

The state estimate is then weighted and accumulated in a binary fashion. In other words, the same element of each sequential pair if sigma points are weighted according to W^m, summed, and stored in shared memory. Then half the threads are deactivated and each pair of the results are summed and stored in shared memory. This continues until all values have been accumulated into a single vector representing the *a priori* state estimate $\hat{x}_{k|k-1}$.

4.4.5 Error Covariance

Calculating the *a priori* error covariance involves equations 2.14 and 3.17 which are repeated for reference.

$$P_{k|k-1} = \sum_{i=0}^{2L} W_i^c [\hat{\chi}_{i,k|k-1} - \hat{x}_{i,k|k-1}][\hat{\chi}_{k|k-1} - \hat{x}_{k|k-1}]^T + Q_k \tag{2.14}$$

$$Q_k = \begin{bmatrix} -q_x & -q_y & -q_z \\ q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \end{bmatrix} \begin{bmatrix} \sigma_{wx}^2 & 0 & 0 \\ 0 & \sigma_{wy}^2 & 0 \\ 0 & 0 & \sigma_{wz}^2 \end{bmatrix} \begin{bmatrix} -q_x & -q_y & -q_z \\ q_w & -q_z & q_y \\ q_z & q_w & -q_x \\ -q_y & q_x & q_w \end{bmatrix}^T \tag{3.17}$$

The value Q_k would be quite cumbersome to obtain through building each of its base matrices and multiplying them together. Instead, the symbolic form of Q_K was solved for and examined for exploitable patterns. When the variance of gyroscope noise on each axis is set equal, two cases appear can be described as:

$$Q_{ij,k} = \begin{cases} -\sigma_w^2 \hat{x}_{i,k-1|k-1} \hat{x}_{j,k-1|k-1} : i \neq j \\ \sigma_w^2 (|\hat{x}_{k-1|k-1}|^2 - \hat{x}_{i,k-1|k-1}^2) : i = j \end{cases} \tag{4.1}$$

This can alternately be represented as

$$\sigma_w^2 \left(I \times \hat{x}_{k-1|k-1}^T \times \hat{x}_{k-1|k-1} - \hat{x}_{k-1|k-1} \times \hat{x}_{k-1|k-1}^T \right) \tag{4.2}$$

where $\hat{x}_{k-1|k-1}$ is a column vector, I is the identity matrix, and σ_w^2 is the noise associated with all gyroscope axes.

A similar approach to the one used with Q_k was taken to clarify the calculation of the summation in equation 2.14. First the state estimate, $\hat{x}_{k|k-1}$, was subtracted from each column of $\hat{\chi}_{k|k-1}$ and the result placed into shared memory. When one thread is assigned

to each result element of the error covariance matrix, their indices determine which two rows of $\hat{\chi}_{k|k-1}$ they multiply together. So, the four by four thread arrangement was used to iterate down the sigma point matrix and weight vector W^c. Each iteration the values from the appropriate two rows and the weight vector were added together and accumulated. The results after completing the iterations were stored back into shared memory. The final computation for the error covariance is shown in Algorithm 4. The Gaussian noise for each axis of rotation is represented by σ_w^2. The variables $\hat{x}_{k-1|k-1}$ and $\hat{x}_{k|k-1}$ represent previous state and state estimate respectively. The center point of the sigma points after propagation through the state transition model are represented as χ_c. The second order weights determined at initialization are represented by W^c. The variable *acc* accumulates the value of a particular result in the error covariance of the estimate matrix, P.

Algorithm 4 Error Covariance

$acc \leftarrow 0$

▷ Find Q

$acc \leftarrow -\sigma_w^2 \times \hat{x}_{k-1|k-1}[i_{4x4}] \times \hat{x}_{k-1|k-1}[j_{4x4}]$
if $i_{4x4} = j_{4x4}$ **then**
  while $itr1 \leq 4$ **do**
    $acc \leftarrow acc + \hat{x}_{k-1|k-1}[itr1]^2$

▷ Find "Center" of sigma points

$\chi_c[i_{2x9}][j_{2x9}] \leftarrow \chi_{k|k-1}[i_{2x9}][j_{2x9}] - \hat{x}_{k|k-1}[i_{2x9}]$
$\chi_c[i_{2x9}+2][j_{2x9}] \leftarrow \chi_{k|k-1}[i_{2x9}+2][j_{2x9}] - \hat{x}_{k|k-1}[i_{2x9}+2]$

▷ Accumulate Error Covariance

while $itr2 \leq 9$ **do**
  $acc \leftarrow acc + W^c[itr2] \times \chi_c[i_{4x4}][itr2] \times \chi_c[j_{4x4}][itr2]$
$P[i_{4x4}][j_{4x4}] \leftarrow acc$

4.4.6 Measurement Model

Equation 3.8 describes the measurement model of the UKF implementation. This is repeated for reference.

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \text{atan2}(2(q_w q_x + q_y q_z, 1 - 2(q_x^2 + q_y^2)) \\ \text{asin}(2(q_w q_y - q_z q_x)) \\ \text{atan2}(2(q_w q_z + q_x q_y, 1 - 2(q_y^2 + q_z^2)) \end{bmatrix} \tag{3.8}$$

Unlike most of the calculations in the UKF, the rows in the result are each calculated substantially different from each other. Though some similarity between the calculations does exist, no easy way was found to create a data access pattern to enable parallel computation between rows. So, for propagating the sigma points through the measurement model, the decision was made to calculate each row in parallel with a set of nine parallel threads. Though code could have been written to attempt to exploit the inter-row calculation similarity, the added complexity of data access would have likely offset most or all of the potential speedup.

When the initial state is near $\pm\pi$ radians in any axis in the measurement space, an additional step must be taken to rotationally correct the results. Otherwise, computational problems arise. Though the points are close in space, the result of the measurement model conversion may place one at $-\pi$ radians and the others at $+\pi$ radians. This discrepancy between reality and the numerical representation causes substantial impact primarily when the difference between predicted heading, $\hat{y}_{k|k-1}$, and measured heading, y_k, is found in equation 2.21. To correct this potential error, the result of the central sigma point, $y_{0,k|k-1}$, is loaded into all threads. If a discrepancy greater than π exists, the output value found by that thread is corrected by a full rotation of 2π in the appropriate direction.

So to calculate the measurement model output, each sigma point is loaded into a thread and normalized to produce accurate Euler angles. Then the thread calculates and stores each element of the result in shared memory. This result is then compared to the central

sigma point and rotationally corrected as appropriate. The results are then accumulated using the same binary approach described at the end of section 4.4.4 to find $\hat{x}_{k|k-1}$.

4.4.7 Matrix Inversion

To find the Kalman gain, K_k, as described in equation 2.19, the inverse of the three by three matrix $P_{\bar{y}_k \bar{y}_k}$ must be found. The standard approach is to set the matrix to be inverted equal to the identity matrix. Then identical row operations are performed on both sides until original matrix is in the form of the identity matrix. The matrix which began as the identity matrix is now the inverse of the original matrix.

The row operations for the inverse process can be described in a standard way. The elements in the first row, or reference row, are all divided by the first element in the row. This produces the first "one" of the final identity matrix. The reference row is then scaled and subtracted from the other rows to produce zeros in the remaining elements of the first column. This series of operations is repeated with each row as the reference row until the inverted matrix is produced.

As the matrix to be inverted is always three by three, the operation can easily be parallelized to initially use all 18 available threads. Each thread is assigned a location within the original matrix or result matrix. The elements of all rows are divided by the first element from its row. Every thread in a row which is not the reference row multiplies its value by the element in its column from the reference row. All of the results are then stored in shared memory and column one's threads are disabled. This is repeated with row two as the reference and then column two is deactivated. Then finally with row three as the reference row. Algorithm 5 describes this full process. The matrix Inv represents the initial matrix concatenated horizontally with the identity matrix to produce a matrix with three rows and six columns. The variable val represents the intermediate results to be stored at each location. The indices i and j represent a mapping of thread IDs to a three by

six arrangement such that an independent thread is assigned a unique location in the Inv matrix.

Algorithm 5 3x3 Matrix Inversion

Ensure: $Inv[3][6] \leftarrow [P_{\bar{y}_k \bar{y}_k}, I]$
  $val \leftarrow Inv[i][j] \div Inv[i][1]$
  if $i > 1$ **then**
    $val \leftarrow Inv[i][j] - val \times Inv[1][j]$
  $Inv[i][j] \leftarrow val$
  if $j \neq 1$ **then**
    $val \leftarrow Inv[i][j] \div Inv[i][2]$
    if $i \neq 2$ **then**
      $val \leftarrow Inv[i][j] - val \times Inv[2][j]$
    $Inv[i][j] \leftarrow val$
    if $j \neq 2$ **then**
      $val \leftarrow Inv[i][j] \div Inv[i][3]$
      if $i \neq 3$ **then**
        $val \leftarrow Inv[i][j] - val \times Inv[3][j]$
      $Inv[i][j] \leftarrow val$

4.4.8 Measurement Noise

The measurement noise matrix, R_k, presents a series of matrix operations in equations 3.20 through 3.24 which appear to provide ample opportunity for data parallelism. To determine if this is actually the case, the elements of R_k were solved for symbolically with the variance noise values assumed to be the same for each axis in a sensor. The results are shown in equations 4.3 and 4.4.

$$\sigma_{\theta\phi}^2 = \frac{(a_z a_y a_x - a_z a_y)\sigma_a^2}{\left(a_z^2 + a_y^2\right)\left(a_z^2 + a_y^2 + a_x^2\right)\sqrt{a_z^2 + a_y^2}} \tag{4.3}$$

$$\sigma_{\psi}^2 = \frac{H_y \sigma_h^2}{H_x^2 + H_y^2}\left(\cos^2(\theta) + \sin^2(\theta) + \sin^2(\phi)\sin^2(\theta) + \sin^2(\phi)\cos^2(\theta) + \cos^2(\phi)\right) \tag{4.4}$$

This reveals that there is little exploitable data parallelism in either of these element's calculations. So, mapping these operations to the GPU would likely result in a loss in

performance. This portion of code was left to the CPU to calculate but was modified from the original series of equations to their reduced form.

4.4.9 Remaining Operations

With the exception of R_k, the remaining equations, 2.17 through 2.21, were solved using the matrix multiplication approach already described. Equations 2.17 and 2.18 were solved using the same approach described in section 4.4.5for error covariance but without the construction of matrix Q_k.

Equation 2.20 was solved for $\hat{x}_{k|k}$ using the matrix multiplication strategy where the right matrix is simply a column vector. Each element of this column vector is first obtained by subtracting the measurement estimate, $y_{k|\hat{k}-1}$, from the rotationally corrected magnetometer heading, y_k. Then the result of the multiplication is added to $\hat{x}_{k|k-1}$ and normalized to find the *a posteriori* state, $\hat{x}_k|k$.

Equation 2.21 is solved by applying the matrix multiplication strategy twice and subtracting the result from $P_{k|k-1}$ to find the *a posteriori* error covariance matrix, $P_{k|k}$.

Next chapter will cover the testing setup used to validate and profile PDR system code. It will examine the results of acceleration and make performance predictions for other platforms more suited for a real-world PDR platform.

Chapter 5

Experimental setup and procedure

This section will describe the testing configurations used and present their relevant intermediate results. First, the data set and the data preprocessing steps are described. Then, the validation process for each PDR component and the system as a whole will be discussed. The chapter concludes with a description of accelerated code testing and a simulation of other platforms of interest.

5.1 Input Data Set

The tests in this chapter were performed using the RuDaCoP[9] data set which was created specifically for development of pedestrian navigation systems. Each trial in this set was collected from a unique participant with sensors worn or carried in a variety of ways. Each participant walks a unique closed loop path in one of several different flat environments. The ground truth data was collected from a pair of foot mounted inertial measurement units (IMUs) and was used to generate the ground truth trajectory information. Each trial is closed loop such that the end point is the same as the starting point. In addition to this ground truth data, two sets of accelerometer, gyroscope, and magnetometer data were collected from smartphones placed in a variety of positions consistent throughout the run. Trials selected for use from this data set were limited to those which are from smartphones carried in a backpack and include trajectory data rated as high accuracy.

Sensor data was preprocessed according to each component's source[1][4]. For step detection and stride length estimation, the gravitational acceleration was subtracted from

the magnitude of acceleration

$$a_t = \sqrt{a_{x,t}^2 + a_{y,t}^2 + a_{z,t}^2} - g \tag{5.1}$$

where $a_{x,t}^2$, $a_{y,t}^2$, and $a_{z,t}$ are the accelerometer readings and g is gravitational acceleration. This was then smoothed with a fourth order Butterworth digital low-pass filter of cutoff frequency 0.2π. For heading detection, five levels of wavelet decomposition were applied to each output of the gyroscope and accelerometer to eliminate high frequency noise[11] for heading estimation.

Magnetometer data was preprocessed to approximate a hard iron calibration. A true hard iron calibration requires rotating the sensor at least one full rotation along each axis. The midpoint of the two outer most calibration readings is then used as a constant offset for every new reading. In these trials, only the two horizontal axes go through a full revolution, so they can be calibrated in two dimensions. The axis orthogonal to the ground was still calibrated in this manner without a full revolution to prevent its value from dominating the rotational correction step of heading estimation.

5.2 System Validation

In order to determine if the PDR system was implemented correctly, the performance of each component was validated followed by the system as a whole. A component was considered validated if it performed above some performance threshold for the majority of randomly selected trials.

Information from the data set regarding ground truth and sensor synchronization was inadequate to ensure perfect synchronization for all components to be tested independently. As such, the trials used for evaluation of SLE and heading are the subset of trials for which step detection detected the same number of steps as the ground truth. A subset of the trials

was constructed from those which met the performance threshold for all three components. The system was considered validated if it performed well in most of this subset of trials.

5.2.1 Step Detection

For each trial used in testing, the SLE algorithm was first trained on the data until convergence. While an ideal case would include unique trials for training and evaluation, each participant only contributes one trial to the data set. This led to a focus on trials including many more steps than the 50 typically required for convergence. A set of 74 trials were selected from runs which contained data from a smartphone placed in a backpack.

Step detection performance was evaluated as the difference between detected steps and steps which appear in the reference. The threshold of $< 1.5\%$ from the ground truth data was chosen. This is represented as

$$\frac{|S_R - S_D|}{S_R} < 1.5\% \tag{5.2}$$

where S_R and S_D are the number of steps in the reference and from the step detection algorithm respectively. The trials which converged make up 66 of the original 74. The percent error of these trials are shown in Table 5.1. This distribution shows that 78% of trials were above the threshold set. As this is a clear majority of the trials, the implementation of the step detection algorithm was considered correct.

The trials which did not meet the threshold were examined for explanations for the higher error. Each of these trials had regions of the acceleration signal which was not consistent with a natural gait or the rest of the signal in the trial. These either occurred often enough to prevent convergence or cause the prediction window to fall on a peak two or more steps ahead. Once this poor prediction occurs, the algorithm is unlikely to recover and tends to predict peaks progressively further into the future.

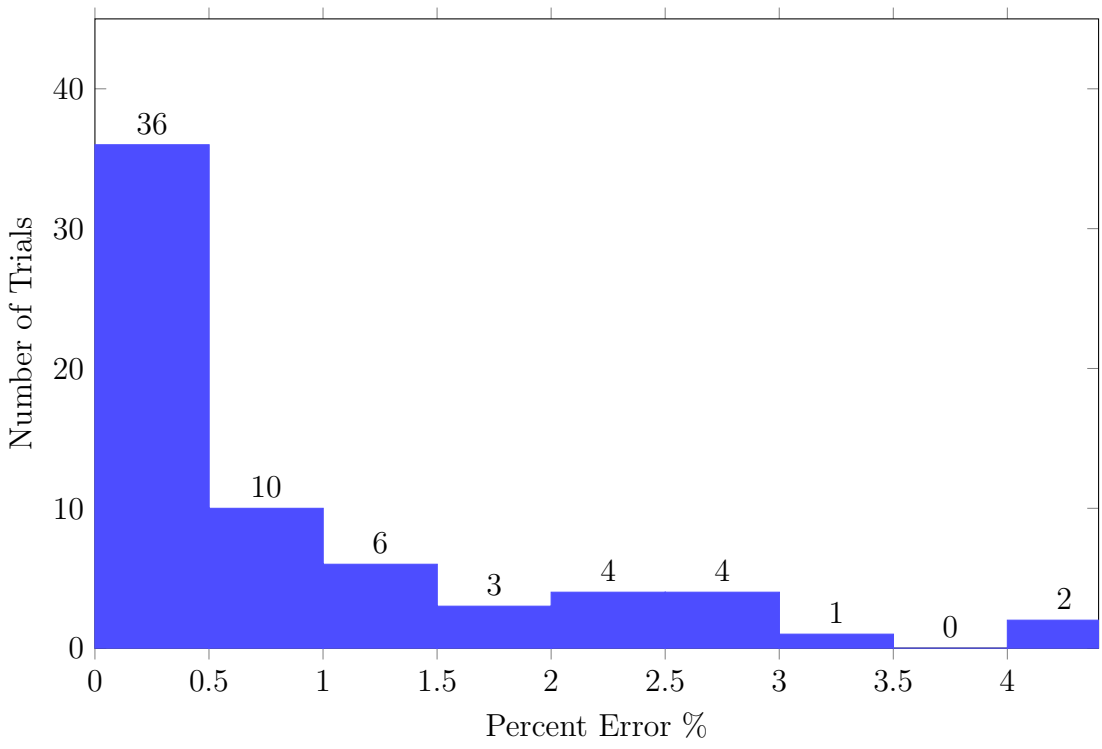

Figure 5.1: Distribution of percent error from 66 step detection trials.

5.2.2 Stride Length Estimation

SLE performance was evaluated as RMS error of the difference in calculated step length and the ground truth position displacement during a step. The error threshold was chosen as < 0.02 RMS error and the convergence criteria chosen as three consecutive steps within 4% of ground truth. This was only calculated for the steps which occurred after the LMS training reached convergence. This can be described as

$$\frac{\sqrt{\sum_{n=1}^{N} P_n^+ - P_n^- - L_n}}{N} < 2\% \tag{5.3}$$

where P_n^- and P_n^+ represent the starting and ending positions of the step, L_n represents the SLE calculation for the step, and N represents the number of steps.

Of the trials used to validate step detection, 19 counted the correct number of steps. These trials are used for SLE validation as they are the most likely to perform as well as ground truth data. The results, shown in Figure 5.2, show that 63% of the trials converged

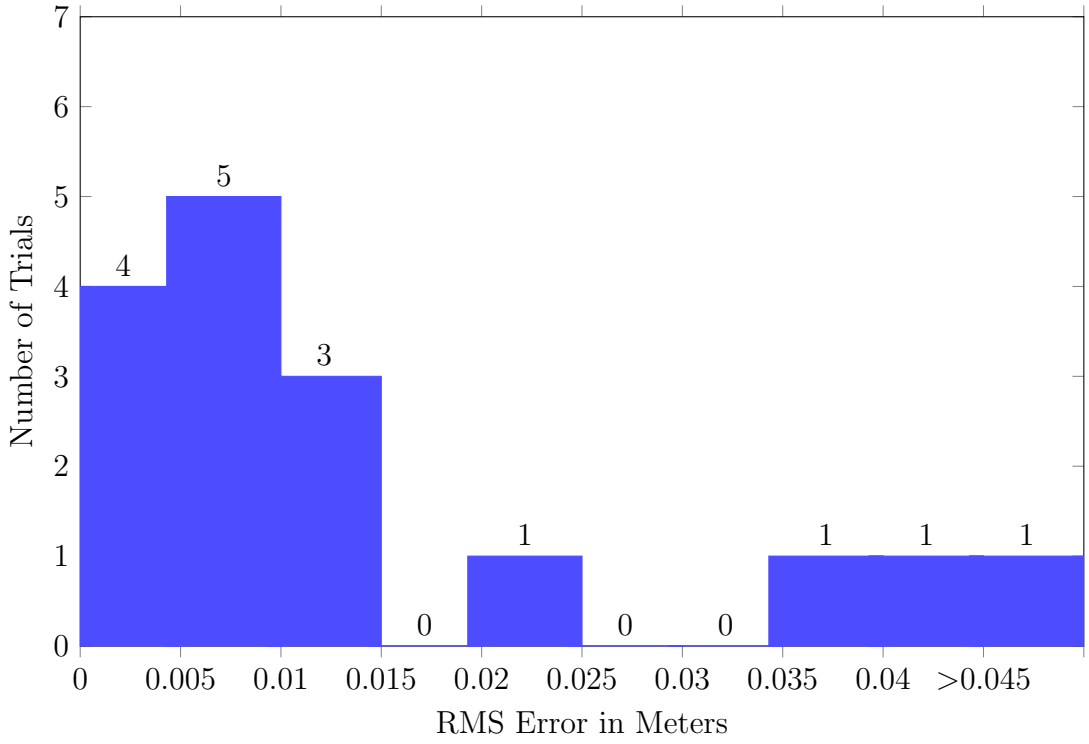

Figure 5.2: Distribution of RMS error from 19 step detection trials. Three trials failed to converge.

within the accuracy threshold. Though the results were less than stellar, they were deemed adequate to prove the algorithm works.

Three of the trials did not converge during training. The error for two of the trials never stabilized which is likely due an erratic walking pattern caused by terrain. These could be rectified with a looser convergence criterion. The third trial began exponential divergence shortly after the trail began. This could potentially be fixed by further tuning the learning rate or introducing a decaying learning rate. However, this was not the primary focus of research so was left for potential future work.

5.2.3 Heading Estimation

Heading estimation performance was evaluated as RMS error between headings averaged over a step and the direction of displacement over the step. This approximation is made to match the implementation from the PDR system. The threshold for correctness was chosen as < 0.03. This can be described as

$$\frac{\sqrt{\sum_{n=1}^{N}\left(\operatorname{atan2}\left(-\Delta P_{y,n}, -\Delta P_{x,n}\right) - \frac{\sum_{i=1}^{H_n} h_i}{H_n}\right)^2}}{N} < 3\% \tag{5.4}$$

where N and n represent the total step count and step number. The values for $\Delta P_{y,n}$, and $\Delta P_{x,n}$ represent the change in y and x position over step n. The number of heading samples in step n is represented by H_n and a particular heading sample is represented by h_i.

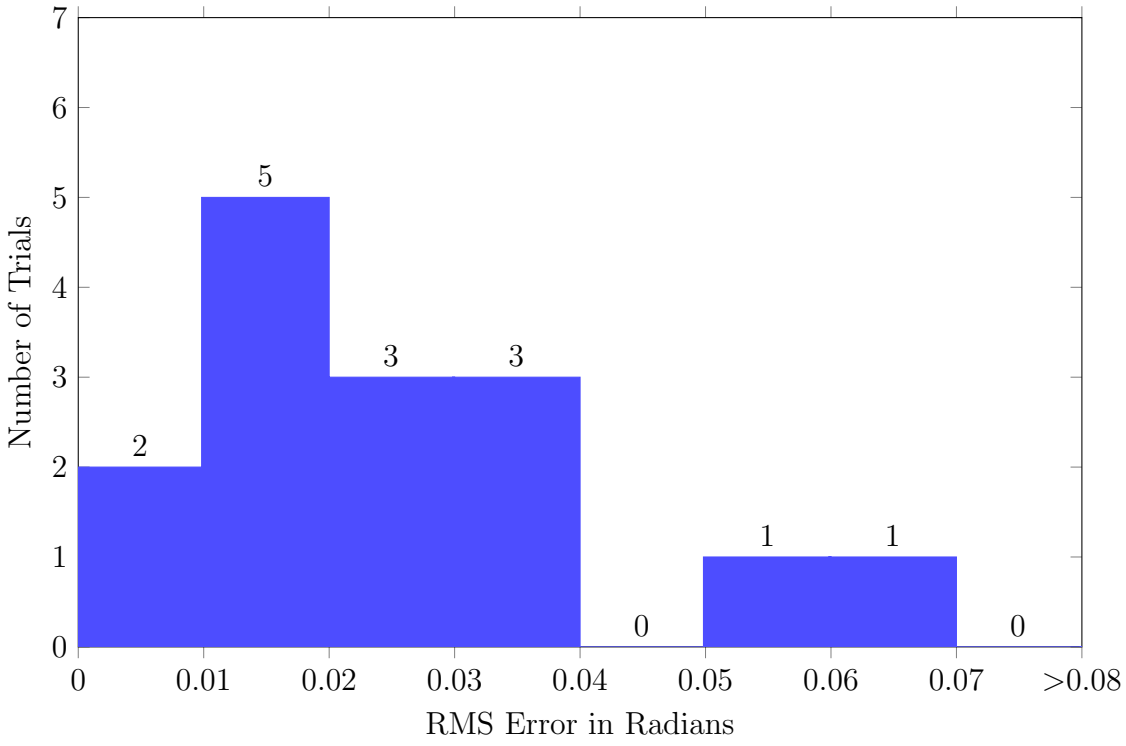

Figure 5.3: RMS heading error distribution from 16 step detection trials. One trial failed to track.

5.3 Acceleration Test Setup

To find the speed-up of the system after acceleration, timing functions were introduced into the code. These timing functions were placed to measure the time spent on each component but exclude the reading from and writing to data files. The timing information for the heading component was collected in two sections. One section includes the magnetometer heading and UKF blocks of Figure 3.5, while the other section surrounds the remaining blocks. The timing block for SLE also includes the portion which handles the system level combination of heading readings and stride length when a step is detected.

In the accelerated form of the code, the UKF block is replaced by code which evaluates to find the measurement noise matrix, R_k, and stores it in memory alongside the other UKF inputs. The timed region for stride length estimation has the system level heading code replaced with the retrieval of the previous heading from memory and queuing of CUDA tasks. The system level code refers to the portion responsible for rotational correction and averaging of UKF heading outputs. This is no longer required as it is handled within the kernel.

The choice to exclude data files in analysis was made because the way data is accessed in a file is different than how it would be accessed in a real system. In a real system, sensor data is always accessed in the same way and takes the same number of clock cycles to retrieve. As mentioned in section 2.3.3, data in memory is moved quite differently. Instead of being moved a single piece at a time, it is moved in varying sized blocks. So, when the sample data is accessed in a file, a large chunk is paged into global memory. Then, a smaller section is moved from there up to the L2 cache from which a small section is moved into L1 cache. Finally, the requested piece of data is moved into a register for use. However, when the next data point is accessed, it likely was already moved into the L1 cache which means a substantially shorter access time. The result is an access time for the data point which is orders of magnitude faster than the previous one. As these two access patterns are quite

different, the timing information from testing would not be analogous to reality and would only hinder analysis of other system components.

Once these modifications were complete, the code was compiled with only the optimization flag, *-O1*, to be consistent with the code used for Callgrind simulations. The compilation did not include the debug flag, *-g*, as it would needlessly slow execution and was accounted for by Callgrind.

The original system and accelerated system were run several times to account for the potential effects of the system running interfering with results. The execution time of the CUDA kernel was found by using NVIDIA's `nvprof` profiling tool with its default settings. The tool simply tracks calls through the CUDA APIs to measure and average execution times.

5.4 Simulation of Other Platforms

This section will discuss the modeling of performance for boards which are more appropriate for our targeted use case than the Jetson Nano. The Jetson Nano is approximately 7 cm by 9 cm and has an order of magnitude greater performance in both CPU and GPU than what is required for real time operation. Because the target use case is something worn on the torso, a device with a much smaller footprint and lower system performance is desirable in a real system.

In order to predict the performance on different systems, a simulation model needs to be generated and validated for accuracy. For this purpose, Callgrind with Cachegrind is used to simulate the Jetson board and its cache system. The relevant results of this simulation are the inclusive clock cycle count estimates for each component of the full PDR system. From this estimate and knowledge of the board's clock frequency, we can produce an estimate of total run time. This can be compared with the actual measured run time. If these results are sufficiently similar, the simulation can then be used to estimate the cycle count on systems with the same or similar processing architecture but different cache arrangements.

Unfortunately, the initial run time calculated for the Jetson was about 10 seconds longer than the actual 36 second run time. In order to generate a model which accurately predicts the Jetson run time from simulation, several changes had to be made to the assumed values which Callgrind provides. The equation used by default to estimate clock cycles is

$$C_{est} = I_r + 10 \times L1_{mr} + 100 \times LL_{mr} \tag{5.5}$$

where C_{est} represents the clock cycle estimate and I_r represents the functions fetched. The miss rate of the level one and last level cache are represented as $L1_{mr}$ and LL_{mr} respectively. The cache miss penalties are assumed to be 10 for the first level cache and 100 for the second by the simulator. Because the number of instructions called and the miss rates are known for each software component, a new set of penalties can be solved for in a system of equations to model the memory system. These can be set equal to an estimate of the clock cycles generated from the times runs and a knowledge of the processor operating frequency.

Clock cycles are estimated by assuming a single instruction is executed for each clock cycle. Immediately apparent when first attempting to solve these equations, is that just the instruction requests are too high to operate as fast as the processor. So, the simplifying assumption was changed in light of how the processor handles pre-emptively loading instructions and that the A57 core can execute more than instructions simultaneously in limited circumstances. Though the details of how this is accomplished are well beyond the scope of this work, a variable representing a scaling factor was added to the I_r in order to solve for an approximation of this behavior. The result is a set of functions of the form

$$C_{est} = \frac{I_r}{x_0} + x_1 \times L1_{mr} + x_2 \times LL_{mr} \tag{5.6}$$

where C_est is now an the experimentally gathered estimate of the clock and x_0, x_1, and x_2 represent the constants to be solved for. As a system of four equations but only three

variables were generated, a small fudge variable, f_0, was added to the equation for SLE to represent system code not associated with it by Callgrind.

A new set of values was solved for which generated an accurate measurement without violating any of the rules implicit from the system being modeled. The variables found are shown in Table 5.1.

x_0	1.3
x_1	15
x_2	150
f_0	$800,000$

Table 5.1: Variables solved for to more accurately model clock cycles.

5.4.1 Platform Selection

The Jetson's processor ARM Cortex A57 based which uses the ARMv8 instruction set architecture (ISA). The ISA is the set of instructions and other attributes the processor is proscribed to implement. Each successive ARM processor which implements the ARMv8 ISA represents some improvements to the performance of the data path. So, the assumption is made that older members of the ARMv8 family of processor will perform worse than or equal to a newer member for a given program. Additionally, the ARMv7 ISA is a reduced version of ARMv8 which only has 13 general purpose registers for computation compared to the ARMv8's 31 registers. This necessitates more time be spent interacting with cache compared to ARMv8 and, consequentially, a decrease in performance if all other things are held equal. For this reason, we consider all ARMv7 processors as generally performing equal to or worse than the Jetson's processor.

These two assumptions of relative performance are important when considering platform selection. When code is run in Cachegrind, it runs as normal on the processor except that Cachegrind intercepts any memory interactions for its own calculations. Because the

way code runs is not modified, the results can be still generalized in light of our relative performance assumptions.

Systems selected for comparison are restricted to those ARM ARMv8 processor which are older than the Jetson's Cortex A57 and those processors which implement ARMv7. These choices only allow for processors whose simulated cycle count will represent a lower bound on the cycles required by the real system. In other words, the real systems are expected to perform worse than the simulations according to the assumptions described. The selected platforms are summarized in Table 5.2. The Pi-Zero W 2 was selected primarily due to its

Board	Frequency	ISA	Core	DataCS	InstrCS	L2 Cache
Jetson	1430 MHz	ARMv8	A57	32 KB	48 KB	2 MB
Pi-Zero W 2	1000 MHz	ARMv8	A53	32 KB	32 KB	512 KB
MVF50NN151CMK40	500 MHz	ARMv7	A5	32 KB	32 KB	-
SVF532R2K2CMK4	400 MHz	ARMv7	A5	32 KB	32 KB	512 KB
Qualcomm 2100 Wear	1000 MHz	ARMv7	A7	16 KB	16 KB	256 KB
STM32MP157C	800 MHz	ARMv7	A7	32 KB	32 KB	256 KB
AM5718xxD	500 MHz	ARMv7	A15	32 KB	32 KB	512 KB

Table 5.2: Platforms Selected for Comparison

small form factor of 6.5x3 cm. This board allows for the creation of a wearable PDR system with both a low weight and size. The Qualcomm 2100 Wear processor was selected as it was designed specifically with wearable devices in mind and has been used in devices such as smart watches. The L1 cache size information is not publicly available, so values were chosen as 16 KB for each cache. This is because the STM32 processor was already simulated as having 32 KB caches for the same core. Additionally, 64 KB is a full quarter of the L2 cache size which makes an unlikely candidate for the designers to choose. The remaining processors were chosen as representations of the types of processors used in internet of things (IoT) applications.

All have small scale SIMD processors with a minimum of four degrees of parallelism in their equivalent of a CUDA block. While the degree of parallelism that can be leveraged

within a block is not as high as in this work, the architectures do allow for easier distribution of threads over multiple thread blocks. In this manner, a set of threads can perform calculation while another waits for data to be available. Additionally, the PDR architecture proposed by this work leverages the ability of a GPU to process independently from a CPU. Due to this, we can expect the same speedup from the perspective of the CPU.

All of the platforms selected were simulated while processing the same extended data trial. The Jetson's performance limit during simulation is its small RAM size which Callgrind overflows quite quickly. In order to performs simulations effectively, the duration of the simulations was limited to 350 steps. While this can be analyzed in a matter of seconds outside of Callgrind, when under simulation the process takes hours. The results of the simulation are then used to construct an estimate of processing time by combining clock frequency of the board and the cycles count from the simulation.

Chapter 6

Results and Discussion

This chapter will discuss the results observed from testing and discuss their implications. The author's recommendations for next steps are listed and discussed. A synopsis of the work done and the conclusions which can be drawn from the data then concludes this work.

6.1 Acceleration Results

The results of independent trials repeated over the same data were combined in Table 6.1 to show impact of the acceleration on the relevant components. The top five trials represent the code before acceleration while the bottom represent acceleration. The results of each set of five trials are combined into an average for easier comparison. The final two rows represent the speedup relative to the original code. The values for speedup are the original execution time divided by the new execution time minus one as shown below.

$$\text{Speedup } \% = \frac{\text{Original Execution Time}}{\text{New Execution time}} - 1 \tag{6.1}$$

Due to the large relative negative impact on SLE from the system level rearranging, the speedup value for its column instead represents a loss of speed. The execution times for step detection are not included as they were nearly identical in each set of trials. Their value is still included in the total execution time column. The percentage found for CUDA speedup is relative to the baseline UKF values. The timing for the Kalman filter is shown but speedup is not as no significant change occurred in its execution time.

The bottom two rows represent the results of speedup calculations performed according to equation 6.1. The second row represents speedups which directly impact the execution

		UKF	CUDA	Kalman Filter	Heading Detect	SLE	UKF SLE	CPU	Total
CPU Only	Trial 1	16.34	-	12.43	28.77	0.0036	16.34	28.87	28.87
	Trial 2	15.8	-	12.26	28.06	0.0034	15.8	28.16	28.16
	Trial 3	16.31	-	12.29	28.6	0.0036	16.31	28.7	28.7
	Trial 4	15.95	-	12.32	28.27	0.0035	15.96	28.37	28.37
	Trial 5	16.55	-	12.58	29.13	0.0035	16.55	29.23	29.23
	Average	16.19	-	12.38	28.57	0.0035	16.19	28.67	28.67
GPU & CPU	Trial 1	2.31	10.6786	12.87	15.19	0.38	2.69	15.64	26.31
	Trial 2	2.26	10.9512	12.62	14.89	0.39	2.66	15.36	26.31
	Trial 3	2.28	10.7384	12.69	14.96	0.38	2.65	15.42	26.16
	Trial 4	2.25	10.758	12.5	14.76	0.37	2.63	15.2	25.96
	Trial 5	2.26	10.6718	12.61	14.87	0.37	2.63	15.32	25.99
	Average	2.27	10.76	12.66	14.93	0.38	2.65	15.39	26.15
Speedup	Theoretical	612%	29%	-	91%	-x107	-	-	10%
	Realized	-	-	-	-	-	511%	86%	-

Table 6.1: Impact of Acceleration on System Performance. All time values listed are for total run time in seconds

time of the CPU and do not depend on the speedups from other components to be fully understood. These are the ones which are most useful to know when determining speedup of this particular system.

The speedup for the SLE and UKF columns are useful for quantifying the cost and benefit of offloading code to the GPU, however, neither can be used effectively to describe the larger system in the absence of the other. In the same way, the speedup for heading detection is interesting to see the scale of the benefit on the particular component, but it is still useless without factoring in the associated costs. The speedup for the CUDA kernel is useful as a metric for determining how effectively the code maps to the GPU but will only impact the CPU directly if it is not completed before another step is detected.

The first pair of columns in table 6.1 represent the effects of acceleration on the UKF. The left represents the speedup within the UKF block from the perspective of the CPU. This is the portion of code which handles the rotational correction of magnetometer data, calculation of R_K, and storing of UKF inputs. The right represents the execution times plus

memory copy times of the CUDA kernel in each trial. However, the speedup value is the old minus the new UKF averages then divided by CUDA average time. These two numbers collectively tell us that code ported to the GPU saw a 29% total speedup, but the CPU saw the speed up as 612% due to the asynchronous computation.

The next two columns describe how the rest of the heading detection algorithm is impacted by this large speedup. The Kalman filter, of course, sees no cost or benefit from the change. its inputs are still the gyroscope and accelerometer, and its output is still to the stripped down UKF code. The impact on the heading detection algorithm as a whole, however, is much more substantial. With the execution time of the UKF cut to nearly a sixth, heading detection as a whole sees an 91% speedup. Unfortunately, this speedup cannot be considered on its own so is largely meaningless.

The third pair of columns describes the interaction between the SLE and UKF components. The SLE algorithm takes a slow down of 107 times its original speed due to its new responsibilities. While this number is quite large, the SLE component initially had the smallest associated cost. In order to determine the actual impact of GPU acceleration, we must combine the slow down to the SLE and speedup from UKF. After combining the initial and final averages from each component before using equation 6.1, we find the true impact of acceleration is a speedup of 511% on the code impacted.

The most important values in this table, however, are the total speedup of the system and the total calculation time. The last two columns describe these in detail. The column labeled CPU tracks the time spent by the CPU in all of the components of PDR. Even though step detection time is not in this table, it is included in the CPU time. The final column describes the total time spent in computation and data transfer for both CPU and GPU. From this we see the system actually spends slightly more time overall than before the acceleration. While this is not ideal, it doesn't mean that nothing was accomplished by performing the acceleration. The 86% speed up seen by the processor frees up clock cycles for it to perform other tasks which cannot be mapped to the GPU.

6.2 Platform Simulation Results

Each row presented in Table 6.2 represents the simulated clock cycle requirements for each PDR component and the predicted execution time of PDR for a particular platform. The first two columns, KF and UKF, represent the clock cycles associated with the Kalman filter and UKF respectively. The next three columns, Heading, StepD, and SLE, represent the clock cycles associated with heading estimation, step detection, and SLE. The last two columns, Total and Predicted Time, represent the total cycle cost of PDR and its predicted execution time.

The clock cycle count for heading is then divided between Kalman filter and UKF columns. The UKF cycles make up the portion of the magnetometer heading and iteration of the UKF from Figure 3.5. The Kalman filter cycles make up the remaining blocks in the figure. The counts for SLE, of course, represent the cost for SLE but they also include the system level costs of combining components to produce a new position.

	Cycles in Billions of Instructions						Seconds
Board	KF	UKF	Heading	StepD	SLE	Total	Predicted Time
Jetson	8.074	11.287	19.362	0.176	0.013	19.55	13.67/13.65
Pi-Zero	8.105	11.236	19.341	0.228	0.013	19.582	19.58
2100 Wear	10.748	14.181	24.928	0.388	0.013	25.329	25.33
STM32MP	9.704	12.566	22.27	0.392	0.013	22.675	28.34
AM5718x	8.604	11.743	20.347	0.202	0.013	20.562	41.12
SVF532R	8.246	11.411	19.657	0.212	0.013	19.882	49.71
MVF50NN	19.731	29.338	49.07	0.705	0.013	49.788	99.58

Table 6.2: The simulation of other platforms' execution time for a three minute data trial. The seconds value for Jetson is its actual run time.

Though most of the platforms do not require PDR acceleration to run real time, they all do see some performance loss compared to the Jetson due to a combination of slower clock rates and smaller cache hierarchies. As mentioned in section 5.4.1, these results likely represent an upper bound on performance as they were generated with their respective cache hierarchies but also have the benefit of the Jetson's better data path.

The processor with the worst performance is the MVF50NN with a predicted execution time of just under two minutes to completely process the three minutes of data in the trial. This board is likely incapable of running just the PDR algorithm in a real scenario. In this situation, proposed system architecture would allow the primary CPU to offload nearly half of the computation requirements of onto its SIMD processor.

For the processors remaining processor, the ability to offload portions or all of a program has several additional benefits beyond those of a standard computing system. As the PDR system seeks to allow for high accuracy personal tracking, there will necessarily be other programs which need CPU time to make the position data useful. The ability to offload a large portion of CPU burden will free up computation time and cache space for other processes to operate concurrently and more efficiently.

6.3 Future Work

There are several changes to the system architecture which could be made to improve system performance. The first would be to change how information collected for a UKF kernel is stored on the CPU. The current arrangement uses an in place queue instead of a rolling one. The result is the SLE estimation must wait for memory to be copied to the GPU before proceeding. It then must copy the content of the buffer forward in memory so at to prevent overflowing out of the back. A rolling buffer would eliminate all of the unnecessary waiting and copying which caused the bulk of the penalties incurred by porting to GPU.

Another clear next step would be to try to offload the Kalman filter to the GPU as well. The basic form of the kalman filter has less data dependency between different equations in a given stage. If arranged carefully, discrete matrix operations could be performed in parallel in a single block. The result could potentially be a GPU block which is operating at near thread capacity for the entire duration of the kernel instead of one which peaks at half like this work's UKF implementation. A carefully designed kernel could even begin work on the next stage before half the result from the previous stage is complete.

Another way the system could see drastic performance improvement is to attempt to interface GPU memory directly with the sensor outputs without the data having to first go to through the CPU. This would allow for kernels which are launched by the CPU and run indefinitely processing data.

One major shortcoming discovered in the approach used for heading detection is performance when the sensing device is not vertical. The magnetometer data is rotationally corrected but the gyroscope data is not. It appears that, when the discrepancy is too large, the UKF heading values seem to exhibit behavior similar to gyroscope drift. An investigation into the source would potentially be beneficial for overall system performance.

As the UKF didn't map as well to the GPU as hoped, an investigation of other data level parallel accelerators would be a reasonable next step. All of the comparison chips simulated also have a SIMD ISA extension called NEON for performing operations with four degrees of parallelism. Some initial work was done in this respect, but time did not allow for substantive progress. Another possibility for investigation would be implementation in an embedded FPGA. This would allow for total control over the flow of data and would likely perform quite well.

An investigation into the impact of hardware acceleration on the simulated MVF processor would be beneficial. It would potentially provide a real-world use case for this work.

6.4 Conclusion

GNSS systems allow for precise geolocation, but their accuracy tends to degrade or disappear in areas where buildings or terrain are tall enough to interfere with the signal. Systems which rely on a precise GNSS reading often need a fallback system when such a signal is unavailable. In order to accomplish this task, sensors must be used to implement a form of tracking until such a time the GNSS signal returns.

PDR technologies seek to provide a solution for pedestrians in such a situation as GNSS is often unreliable in cities and hilly terrain. These are usually on embedded systems with

power and computation resources. Leveraging computational parallelism can ease the strain on a system and decrease power consumption.

The primary goal of this work was to assess the applicability of GPU acceleration to a PDR system. To that end, a full PDR system was built using representative approaches, validated for correctness, and profiled. The subsystem for determining heading uses a UKF for fusion of discrete headings into a single one with a higher degree of accuracy. This component took up half of the computation resources of the system and exhibited a high degree of parallelism

The UKF was rewritten in CUDA for the Jetson Nano's architecture to leverage the parallelism present in computation with some success. The system structure was then rearranged to minimize overheads associated with GPU acceleration and decouple the UKF as much as possible from the rest of the system.

The unmodified and accelerated systems were both analyzed with respect to the amount of time spent on computation and the results compared. The system managed to achieve a 86% system level speedup on the CPU but the total duration of computation remained essentially unchanged. This showed that enough parallelism exists within a four dimensional UKF to allow for an equivalent execution time on a GPU but not enough to see a benefit or loss in reducing consumed processing time for the target board.

A sampling of other processors and boards with fewer resources were simulated to predict their performance in running the program design implemented. Most of the systems are predicted to be able to run with a comfortable margin, but one particular case was discussed which may become capable of meeting real time criteria only after the proposed hardware acceleration.

Bibliography

[1] Y. Yao, L. Pan, W. Fen, X. Xu, X. Liang and X. Xu, "A Robust Step Detection and Stride Length Estimation for Pedestrian Dead Reckoning Using a Smartphone," in IEEE Sensors Journal, vol. 20, no. 17, pp. 9685-9697, 1 Sept.1, 2020, doi: 10.1109/JSEN.2020.2989865.

[2] Z. Xiao, H. Wen, A. Markham and N. Trigoni, "Robust pedestrian dead reckoning (R-PDR) for arbitrary mobile device placement," 2014 International Conference on Indoor Positioning and Indoor Navigation (IPIN), 2014, pp. 187-196, doi: 10.1109/IPIN.2014.7275483.

[3] A. Wang, X. and B. Wang, "Improved Step Detection and Step Length Estimation Based on Pedestrian Dead Reckoning," 2019 IEEE 6th International Symposium on Electromagnetic Compatibility (ISEMC), 2019, pp. 1-4, doi: 10.1109/ISEMC48616.2019.8986071.

[4] A. Poulose, B. Senouci and D. S. Han, "Performance Analysis of Sensor Fusion Techniques for Heading Estimation Using Smartphone Sensors," in IEEE Sensors Journal, vol. 19, no. 24, pp. 12369-11070, 15 Dec.15, 2019, doi: 10.1109/JSEN.2019.2940071.

[5] W. Beibei, C. Tao and Z. Zhao, "An Improved in Stride Estimation Algorithm of Pedestrian Dead Reckoning," 2017 9th International Conference on Measuring Technology and Mechatronics Automation (ICMTMA), 2017, pp. 154-157, doi: 10.1109/ICMTMA.2017.0046.

[6] K. Lan and W. Shih, "Using simple harmonic motion to estimate walking distance for waist-mounted PDR," 2012 IEEE Wireless Communications and Networking Conference (WCNC), 2012, pp. 2445-2450, doi: 10.1109/WCNC.2012.6214207.

[7] N. Strozzi, F. Parisi and G. Ferrari, "A Novel Step Detection and Step Length Estimation Algorithm for Hand-held Smartphones," 2018 International Conference on Indoor Positioning and Indoor Navigation (IPIN), 2018, pp. 1-7, doi: 10.1109/IPIN.2018.8533807.

[8] Yang, Depeng & Sun, Junqing & Lee, JunKu & Liang, Getao & Jenkins, David & Peterson, Gregory & Li, Husheng. (2010). Performance Comparison of Cholesky Decomposition on GPUs and FPGAs.

[9] A. Bayev, I. Chistyakov, A. Derevyankin, I. Gartseev, A. Nikulin and M. Pikhletsky, "RuDaCoP: The Dataset for Smartphone-based Intellectual Pedestrian Navigation,"

2019 International Conference on Indoor Positioning and Indoor Navigation (IPIN), 2019, pp. 1-8, doi: 10.1109/IPIN.2019.8911823.

[10] B. Shin et al., "Indoor 3D pedestrian tracking algorithm based on PDR using smarthphone," 2012 12th International Conference on Control, Automation and Systems, 2012, pp. 1442-1445.

[11] Vadim Bistrov, "Performance Analysis of Alignment Process of MEMS IMU", International Journal of Navigation and Observation, vol. 2012, Article ID 731530, 11 pages, 2012. https://doi.org/10.1155/2012/731530

[12] M. N. Muhammad, Z. Salcic and K. I. Wang, "Real-time PDR based on resource-constrained embedded platform," 2015 9th International Conference on Sensing Technology (ICST), 2015, pp. 779-784, doi: 10.1109/ICSensT.2015.7438502.

[13] S. Urmat and M. E. Yalçın, "Design and implementation of an ARM based embedded system for pedestrian dead reckoning," 2015 9th International Conference on Electrical and Electronics Engineering (ELECO), 2015, pp. 885-889, doi: 10.1109/ELECO.2015.7394520.

[14] T. Fukagai et al., "Speed-Up of Object Detection Neural Network with GPU," 2018 25th IEEE International Conference on Image Processing (ICIP), 2018, pp. 301-305, doi: 10.1109/ICIP.2018.8451814.

[15] N. singh and S. P. Panda, "Enhancing the Proficiency of Artificial Neural Network on Prediction with GPU," 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon), 2019, pp. 67-71, doi: 10.1109/COMITCon.2019.8862440.

[16] L. Liu, J. Luo, X. Deng and S. Li, "FPGA-based Acceleration of Deep Neural Networks Using High Level Method," 2015 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2015, pp. 824-827, doi: 10.1109/3PGCIC.2015.103.

[17] T. Xiao and M. Tao, "Research on FPGA Based Convolutional Neural Network Acceleration Method," 2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA), 2021, pp. 289-292, doi: 10.1109/ICAICA52286.2021.9498022.

[18] W. Zhang, D. Wei, H. Yuan and G. Yang, "Cooperative Positioning Method of Dual Foot-Mounted Inertial Pedestrian Dead Reckoning Systems," in IEEE Transactions on Instrumentation and Measurement, vol. 70, pp. 1-14, 2021, Art no. 8502114, doi: 10.1109/TIM.2021.3066173.

[19] M. Zhou, Y. Wei, Z. Tian, X. Yang and L. Li, "Achieving Cost-Efficient Indoor Fingerprint Localization on WLAN Platform: A Hypothetical Test Approach," in IEEE Access, vol. 5, pp. 15865-15874, 2017, doi: 10.1109/ACCESS.2017.2737651.

[20] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020.

[21] Harris, M. (2020, August 25). Maxwell: The most advanced CUDA GPU ever made. NVIDIA Technical Blog. Retrieved April 22, 2022, from https://developer.nvidia.com/blog/maxwell-most-advanced-cuda-gpu-ever-made/

[22] DATA SHEET NVIDIA Tegra X1 Series Processors. Nvidia Developer. (2015, November). Retrieved April 22, 2022, from https://developer.download.nvidia.com/assets/embedded/secure/jetson/TX1/docs/TegraX1_Embedd
DkwVcvPnseQcFBgvjTS12w0BLTBFZj7oQw1J94VNWN3tvn-
8qV6uqVDu8TWTgXeKAEOoDp9DwB0DuKo79onvosZ0sPxtsFxAoHkJotTnSYhy9achqVyK1wUIC

[23] Valgrind user manual. Valgrind. (n.d.). Retrieved April 11, 2022, from https://valgrind.org/docs/manual/manual.html

[24] Nethercote, N., Walsh, R., & Fitzhardinge, J. (2006, October). IISWC-2006 tutorial building workload ... - valgrind. Valgrind. Retrieved April 11, 2022, from https://valgrind.org/docs/iiswc2006.pdf

[25] Cachegrind user manual. Valgrind. (n.d.). Retrieved April 11, 2022, from https://valgrind.org/docs/manual/cg-manual.html

[26] Callgrind user manual. Valgrind. (n.d.). Retrieved April 11, 2022, from https://valgrind.org/docs/manual/cl-manual.html

[27] Segal, M., Akeley, K. (2019, October 22). OpenGL 4.6 (core profile) - october 22, 2019 - Khronos Group. Khronos.org. Retrieved April 18, 2022, from https://www.khronos.org/registry/OpenGL/specs/gl/glspec46.core.pdf

[28] RadeonOpenCompute. (2021, December). ROCm_Documentation/hip-guide.rst at master · RadeonOpenCompute/rocm_documentation. GitHub. Retrieved April 18, 2022, from https://github.com/RadeonOpenCompute/ROCm_Documentation/blob/master/Programming_Guides/HIP-GUIDE.rst