UCGE Reports

Number 20306

Department of Geomatics Engineering

Co-Processor Aiding for Real-Time Software GNSS Receivers

(URL: http://www.geomatics.ucalgary.ca/graduatetheses)

by

Aleksandar Knežević

May 2010

UNIVERSITY OF CALGARY

Co-Processor Aiding for Real-Time Software GNSS Receivers

by

Aleksandar Knežević

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTERS OF SCIENCE

GEOMATICS ENGINEERING

CALGARY, ALBERTA

FEBRUARY 2010

Abstract

Increasing interest exists in replacing hardware components of GNSS receivers with software. Moving the digitization closer to the antenna has the major benefit of added flexibility. Algorithms can be developed and tested effectively and in a timely manner. Implementations can be modified with a simple software update rather than a hardware change. General purpose radios can be used for GNSS, requiring only specific software, not necessarily hardware.

The major drawback, however, is the need for large amounts of processing power to perform Doppler removal and correlation (DRC). This is especially true with the introduction of new, higher bandwidth signals which require sampling rates upwards of 40 Msps. With each channel being tracked requiring six multiplications and four additions per sample per code phase, in order to achieve real-time operation, it becomes necessary to offload some of the processing from the central processing unit (CPU) onto a coprocessor.

To the best of the author's knowledge, the only functional Graphics Processing Unit (GPU) aided GNSS receiver in development is in the PLAN group (Petovello et al 2008). The receiver is based on GSNRx™ – the PLAN group's software GNSS receiver – and is capable of performing DRC processing for eight satellites on 1 ms of 25 Msps data in less than 1 ms, suggesting real-time capability, although no real-time capability had

previously been implemented. This research starts with a complete re-design of the DRC module and extends the real-time capability of GSNRx™ to 40 Msps.

The new stand-alone DRC module for the GPS L1 C/A signal described herein has been developed using the NVIDIA CUDA software development kit. The module employs a multi-threaded asynchronous design and as such utilizes CPU resources only when initiating correlation tasks. All DRC operations are performed on an NVIDIA GeForce 8800 GTX.

The DRC module has been integrated into GSNRx™. Timing as well as tracking and navigation solution results are given.

Acknowledgements

I could not have completed this thesis without the support of many people. I would like to express my gratitude to a few of them now.

- My supervisor Dr. Gérard Lachapelle for his support of my research and not only for being a great role model, but for always having an open door.

- My co-supervisor Dr. Cillian O'Driscoll for his great ideas, patience and support throughout my work. Working with Cillian has been a great experience which has made me a better researcher and a better engineer.

- All the friends I've made during my stay in Calgary. You have greatly contributed to making these last two years a fantastic part of my life.

- All of my colleagues at the Geomatics department for being there and always offering a helping hand when one was needed.

Lastly, I would like to thank everyone else who was important in the realization of this thesis. I am sorry I could not mention you all.

Table of Contents

List of Tables

List of Figures and Illustrations

List Abbreviations

Symbol	Definition
ALU	Arithmetic Logic Unit
API	Application Programming Interface
BLE	Basic Logic Element
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DMA	Direct Memory Access
DRC	Doppler Removal and Correlation
DSP	Digital Signal Processing
FIFO	First-In First-Out
FLOPS	Floating Point Operations Per Second
FPGA	Field Programmable Gate Array
GPGPU	General Programming on GPUs
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language
IF	Intermediate Frequency
LAN	Local Area Network
LBS	Location Based Services
LUT	Look-Up Table
NCO	Numerically Controlled Oscillator
PRN	Pseudo Random Noise
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
sps	Samples per second

Introduction

There are two major forces driving software GNSS receivers: cost / integrateability and their use as a research platform.

Increasing demand for Location Based Services (LBS) has led to the increasing popularity of GNSS receivers being incorporated into other platforms, such as mobile phones. Traditionally, this is performed by having two completely separate antennas / signal paths, one for communication and the other for GNSS. By moving the digitization of the signals closer to the antenna, software receivers give system designers the ability to share resources between general purpose radios and GNSS, thus reducing cost.

General purpose radios can be used for GNSS, requiring only specific software, not necessarily hardware. The major drawback, however, is the need for large amounts of processing power.

With the increase in the number of new GNSS signals, it is important to have suitable development platforms to conduct this research.

Using a software based receiver, algorithms can be developed and tested quickly by researchers without the need for specialized hardware or training. Implementations can be modified with a simple software update rather than a hardware change. Again, the major

drawback is the need for large processing power required for real-time operation or the large storage required for post-processing.

This thesis will present two major contributions: the work performed to decrease the processing time required for high-bandwidth GNSS signals by offloading tasks to aiding hardware, and the work performed to allow real-time processing of GNSS signals.

1.1 Limitations of Previous Work

There has been considerable success in the development of software GNSS receivers both in the PLAN group, and outside (Borre 2007, Angheileri 2007, Pany 2004, Ledvina 2003, Schamus 2002, Thor 2002), however work has been limited to either low sample rate or post mission processing. To the best of the author's knowledge, there is no PC based software receiver available which can process high sample rate data – 25 mega samples per second (Msps) or more – on eight satellites in real-time.

For the purpose of this work, PC based software receivers can be divided into three categories: pure software, Field Programmable Gate Array (FPGA) aided and Graphics Processing Unit (GPU) aided. The following is a brief description of existing work in each category.

1.1.1 Pure Software GNSS Receivers

During the last few years, many real-time PC-based software GNSS receivers have been developed (Borre 2007, Angheileri 2007, Pany 2004, Ledvina 2003, Schamus 2002, Thor 2002, Petovello & Lachapelle 2008). All are able to track GPS L1 C/A on at least four satellites at a sample rate of 5 MHz.

ipexSR (Pany 2004) stands out as the only pure software receiver capable of tracking high bandwidth signals. It is able to process up to 13 correlator pairs at a 33.3 Msps sampling rate on a 3.0 GHz PC. This equates to tracking the early prompt and late replicas of four satellites. Time critical functions have been implemented in assembly to decrease overhead and increase computational efficiency.

GSNRx™ (Petovello & Lachapelle 2008), developed by the PLAN group, is capable of tracking up to eight satellites at a 5 Msps sampling rate in nearly double real-time on a 1.6 GHz PC. A multi-threaded, multi-processor implementation has been developed and is currently being tested. This version is capable of real-time operation and utilizes some of the software developed during this research.

In order to obtain such high processing rates, both implementations utilize the Intel chipsets single instruction multiple data (SIMD) functions (Charkhandeh 2007). These functions allow the processor to perform an identical arithmetic operation on 16

individual data points, assuming 16-bit data, in one clock cycle. The limitation of this approach will always be the width of a General Purpose Processor's (GPP's) Arithmetic Logic Unit (ALU), and the clock speed. As current implementations are already nearly fully utilizing the available hardware, with present hardware it would be very difficult to increase the sampling rate.

1.1.2 FPGA Aided Software GNSS Receivers

To the best of the author's knowledge, no work has been published on a PC based FPGA assisted GNSS receiver.

However, two labs have been developing Digital Signal Processing (DSP) / FPGA based GNSS receivers, namely SNAP at the University of New South Wales (Engel et al 2006) and the Navigation Lab at the Politecnico di Torino (Dovis 2005).

Both labs have developed a similar architecture in which an FPGA is used to take in IF samples, perform Doppler Removal and Correlation (DRC) and pass data onto a DSP which performs the higher level functions. Both receivers are intended to be stand-alone boards.

There are two drawbacks to the FPGA approach. First, hardware is specialized and unavailable on most PCs and second the implementation is more difficult to maintain and upgrade as fewer researchers in the field are familiar with HDL than with C.

1.1.3 GPU Aided Software GNSS Receivers

As of the outset of this thesis, the only GPU aided GNSS receiver in development is in the PLAN group (Petovello et al 2008). The receiver is based on GSNRx™ and is capable of performing Doppler Removal and Correlation (DRC) processing for eight satellites on 1 ms of 25 Msps data in less than 1 ms, suggesting real-time capability. No real-time capability had been implemented at the start of this research.

During the course of this work, the National Institute of Information and Communications Technology in Japan had developed a GPU aided software receiver (Hobiger et al 2009). It does not use traditional early, prompt, late correlator scheme for the code tracking loop, rather it exploits the fast FFT functionality of the GPU to create a finer correlation function. The receiver is real-time capable up to 16 Msps.

1.2 Objectives and Contributions

The overall objective of this thesis is to develop and test a co-processor aided, real-time, PC based, software GNSS receiver. The starting point is the GNSS Software Navigation

Receiver (GSNRx™), a C++ class-based software receiver developed by the PLAN group, capable of processing raw data samples from a GNSS front-end (Petovello & O'Driscoll 2007). Since the biggest challenge associated in a real-time functionality is the processing power required by the DRC and signal generation (Petovello et al 2008), this research will concentrate on offloading these functions to a GPU. In order to maintain flexibility in the software, the goal will be to keep as many operations as possible in a high level language on the PC. To keep the system general, signal type, code ID and other parameters will be transparent to co-processor tasks.

This objective can be divided into the following tasks:

- Implementation of a stand-alone DRC kernel on GPU hardware
- Integration of the DRC into GSNRx™
- Modification of tracking / measurement algorithms to accommodate the new correlation structure
- Incorporation of a real-time front-end into the existing software
- Real-time testing and optimization

1.3 Thesis Outline

The subsequent chapters are structured as follows.

Chapter 2 reviews the theoretical background of GNSS receivers, concentrating on Doppler removal and correlation. Special interest is paid to software based GNSS receivers, specifically to the processing load required for Doppler removal and correlation (DRC). Following this is a discussion on different processor technologies including general purpose processors (GPPs), digital signal processors (DSPs), graphic processing units (GPUs) and field-programmable gate arrays (FPGAs). The chapter concludes with a discussion on algorithm parallelization.

Chapter 3 breaks down the contributions of this work which were made in order to create a real-time co-processor aided receiver. First, streams which had to be abandoned are discussed. This includes the DRC design for a Xilinx FPGA and for an ATI GPU. Following this, the design cycle for the DRC module on an NVIDIA GPU is outlined. Design challenges and their solutions are presented. The chapter concludes with a description of the final DRC module.

Chapter 4 outlines the design of the separate sample source modules, created for this work, required for real-time operation. The chapter concludes with an overview of the real-time receiver, and contrasts it with the post-processing receiver.

Chapter 5 provides results of the various tests performed to ensure stable real-time operation. It starts with raw processing rates of the DRC module with different optimization specifications. This section includes the processing times required for each

kernel call. Following this, the sample source – the hardware capable of streaming GNSS samples to the processing PC – is analyzed to show transfer rate capability. The chapter concludes by showing resource usage – CPU and network – for different sampling rates in real-time operation.

Chapter 6 draws conclusions from this work. Suggestions are made for further research.

Chapter Two: Background

This chapter starts off with a general overview of GNSS receiver structure and then details specifics about software GNSS receivers. Following this, an overview of available processor technologies is presented. Finally, the chapter is concluded with a discussion on parallel processing.

2.1 GNSS Receiver Overview

A GNSS receiver can be divided into several modules: the front end, which is composed of an antenna, a frequency synthesizer, a down-converter and a digitizer, multiple channels, which are composed of a Doppler Removal and Correlation (DRC) module and a tracking loop, and finally, a navigation solution estimator.

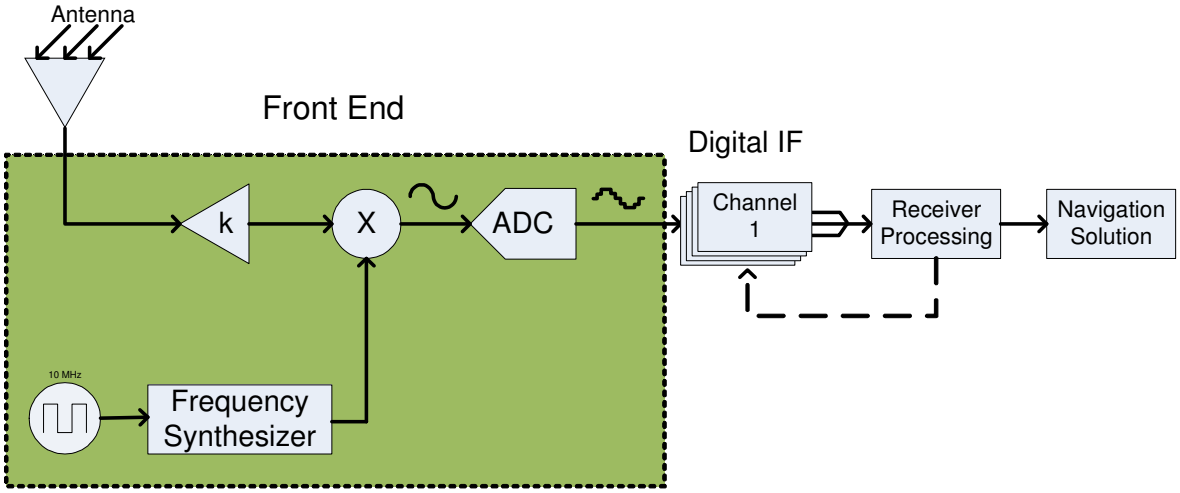

Figure 1 -- GNSS Receiver Overview

Figure 1 shows an overview of a general receiver. After amplification from the antenna, signals are down-converted to an intermediate frequency (IF) and digitized. Multiple parallel channels then process the digitized samples to remove the IF and Doppler, thus bringing them to baseband, and to despread their spectrum. More details on this follow. A discriminator evaluates each channel to produce an error estimate between the incoming signal and the locally generated replica. This value is then filtered by the tracking loop filter and fed back to the DRC module. At measurement epochs, relative propagation delay measurements are generated from each signal and used to produce a navigation solution estimate.

For brevity, functional details about the front end, the tracking loops and the navigation solution estimator will be omitted from this discussion. The reader is referred to Ward (2005) Chapter 5, for an in-depth study.

The work presented in Chapter 3 is based on parallelization and optimization of the DRC algorithm, which is described in further detail here.

After down-conversion and digitization, assuming real sampling, a constant data bit, as integration is contained within one bit, and ignoring noise, the signal at the input of the DRC module can be represented as a sum of signals, where the i^{th} signal can be represented as:

$$r_i[n] = d_i c_i[nT_s - \tau_i]\cos(2\pi\{f_{IF} + f_{di}\}nT_s + \theta_i) \tag{2.1}$$

where:

i is the satellite number.

n is the sample number,

d_i is the navigation message data bit,

c_i is the spreading sequence chip,

τ_i is the spreading sequence code phase,

f_{IF} is the intermediate frequency of the down-converter,

f_d is the Doppler frequency caused by user and satellite motion,

T_s is the sampling period and,

θ_i the phase of the incoming signal.

In order to be able to recover the navigation message data sent by the satellite, the receiver needs to reproduce a local replica of the incoming signal for two reasons: first, due to satellite and user motion, and clock imperfections, the incoming signal frequency is not known and cannot be down-converted to baseband by hardware and second, due to the spreading sequence, the signal power is spread over a much larger frequency band and is thus well below the noise floor (Lachapelle 2008).

The spreading sequence is a high frequency pseudo-random noise (PRN) code which is XORed with the navigation data at the satellite. This is done for three reasons. First, the high rate sequence acts to spread the signal power over a larger frequency range – see Figure 2. This makes the signal less prone to interference in the transmission path.

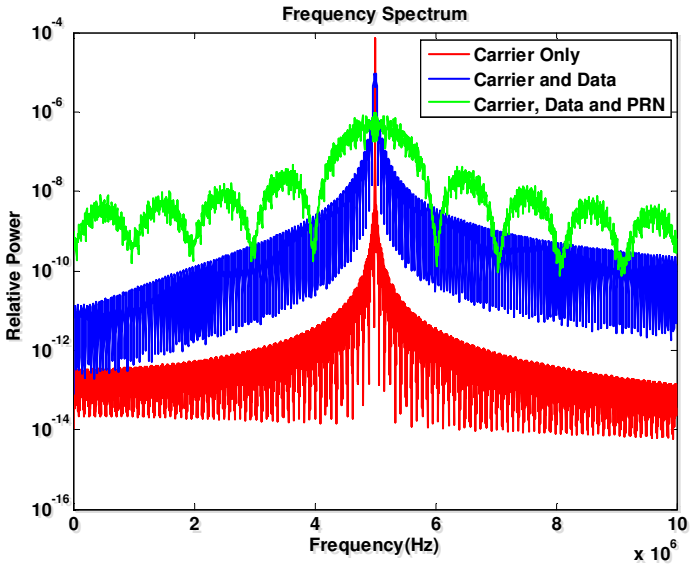

Figure 2 -- Frequency Spectrum of CDMA Signal

Second, by modulating each signal with a different spreading code, separation is provided. That is, multiple signals can be transmitted on the same carrier frequency at the same time and each can be recovered using its specific code (Viterbi 1995). Third, and most important in the field of navigation, by modulating the signal with a deterministic code, the user can obtain relative transmission delay information between signals from the incoming code phase.

Figure 3 shows the details of a DRC module.

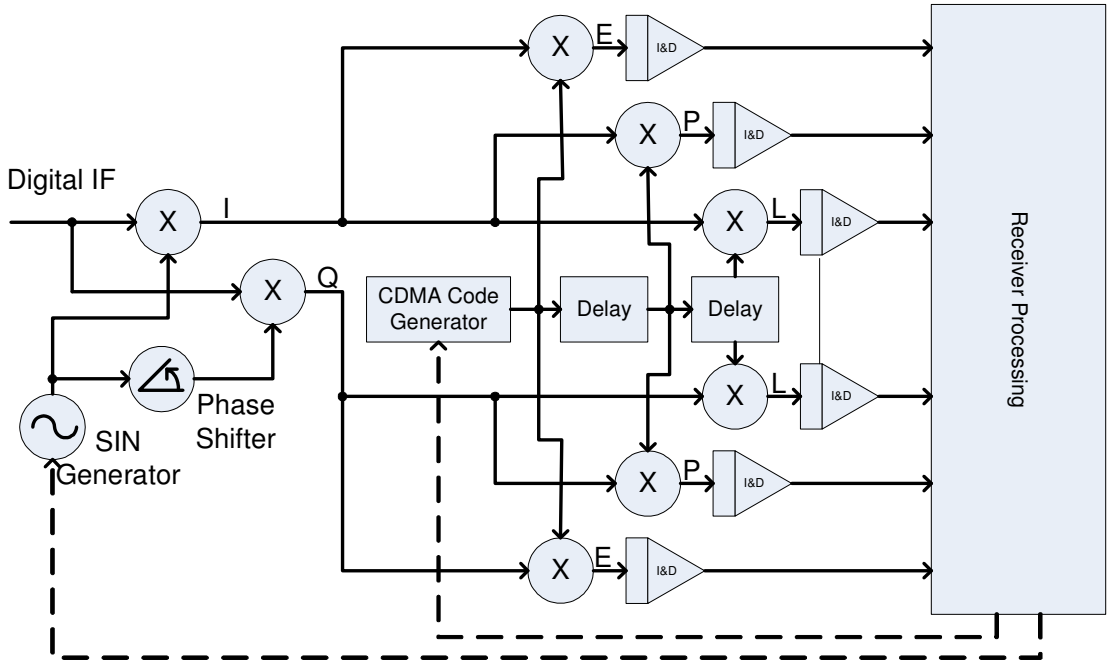

Figure 3 -- Single Channel DRC

The digitized samples are first multiplied by a complex local carrier replica, which brings them to baseband. After this, the two components are multiplied by delayed versions of

the spreading sequence: an early, a prompt and a late replica. The results for each are summed up producing the following output, again assuming a constant data bit:

$$C = dR_s(\Delta\tau)\frac{\sin(\pi\Delta fNT_s)}{N\sin(\pi\Delta fT_s)}\exp\{j\pi f(N-1)T_s + j\Delta\theta\} \tag{2.2}$$

where:

d is the navigation message bit,

R_s is the correlation function at a code offset $\Delta\tau$,

Δf is the error in the frequency of the locally generated carrier replica, and $\Delta\theta$ is the error in the phase of the locally generate carrier replica.

The results from this correlation are passed through a series of discriminators to estimate $\Delta\tau$, Δf and $\Delta\theta$, and tracking parameters are updated for the next correlation epoch.

2.2 Software GNSS Receiver

A software receiver will implement all tasks described in the previous section, following sampling, in software. This section will begin with a brief history of software receivers, followed by an overview of the challenges faced in software receiver design, and finally, conclude with some methods developed to mitigate those challenges.

2.2.1 History of Software GNSS Receivers

Like a lot of technology, software GNSS receivers, and software defined radio in general was driven by military development. In the early 1990s, U.S. military services were utilizing radios with dedicated hardware components optimized for specific field applications (Won et al 2006). The design life of these radios was typically assumed to be 30 years. At the same time, commercial radio applications began driving up the pace of technology development such that the effective lifetime of any given component design was two years.

To accommodate the rapid change in equipment design, the United States Department of Defense initiated a project called *Speakeasy* as a proof of concept for a programmable waveform, multiband, multimode radio (Lackey & Upmal, 1995). The developed approach was software defined radio (SDR): the digitizer is placed as close as possible to the antenna, thus IF data is processed using software techniques rather than by dedicated hardware.

The software implementation of all baseband functions facilitates easy design modification and updates. This flexibility was very beneficial in a military setting which required communication at different frequencies, modulation types, spreading sequences and baseband algorithms.

Over the years, several advancements in SDR research have made real-time software GNSS receivers possible. One such advancement came in 1990, when NASA and Caltech JPL introduced an FFT based acquisition scheme for CDMA signals. This was later improved upon by (Nee & Coenen 1991) to allow for FFT and IFFT based acquisition of GPS signals.

Work on the GNNS software receiver began with Akos (1997) with the development of the first receiver to rely solely on software algorithms for signal acquisition and tracking. Since then, multiple centers have been researching the topic, see section 1.1. Until recently, all software GNSS receivers recorded data, and processed it post-mission. (Akos et al2001) were first to develop and present a real-time software GNSS receiver. A large amount of current research interest is focused on real-time capable receivers.

2.2.2 Challenges in Software GNSS Receiver Design

As alluded to in Chapter one, the biggest challenge of software receivers is their high computational requirement. To that end, the required processing tasks are divided into high-rate, medium-rate and low-rate operations (Petovello et al 2008).

2.2.2.1 High-Rate Operations

High rate operations are performed at the sampling rate of the receiver, a minimum of 4 MHz for GPS L1 C/A and up to 25 MHz for other signals assuming real sampling. These include DRC and signal generation. In order to produce three delayed outputs for tracking, DRC requires at least eight multiplications and six additions per sample per signal (Petovello & Lachapelle 2008). Signal generation requires one sine / cosine generation and three ranging code sample generations per sample per signal.

The high rate operations are the greatest computational load on the receiver, and as such, are the focus of this work.

2.2.2.2 Medium-Rate Operations

Medium rate operations are preformed after every correlation epoch, between 50 Hz and 1 kHz. That is, these tasks are performed on the output of the high-rate tasks. Operations include discriminators, tracking loop updates, and navigation message extraction.

2.2.2.3 Low-Rate Operations

Low rate operations are performed at the output rate of the receiver, between 100 Hz and 1 Hz. These operations include measurement generation and navigation solution computation and are performed independently of high and medium-rate operations.

Table 1 summarizes how many mathematical operations per second (additions or multiplications) are required per channel to perform only the high-rate tasks mentioned above. The disparity in calculation requirement is due to the difference in bandwidth of the signals. Higher bandwidth signals need to be sampled at higher rates, thus require more calculations per second.

Table 1 – Operations per Second

Signal	Operations / Second
GPS L1 C/A	320 M
Galileo E1	640 M
Galileo E5A	3.2 G

2.2.3 Methods for Reducing Computation Cost in Software GNSS Receivers

In a serial software receiver, two methods are usually implemented in order to reduce computational cost. First signal and code generation are usually implemented by a look-up table, rather than direct computation (Ledvina et al 2002).

Since storing every possible carrier frequency would be unfeasible, a coarse grid is used. The introduced difference in Doppler is later removed by a phase shift. Even so, the downside of this approach is the large memory requirement. If a spacing of 50 Hz is used,

400 look-up tables are needed for a +/- 10 kHz Doppler range. This means that at a sampling frequency of 20 MHz, 16 Msamples of memory are needed. That is, 16 MB and 32 MB of memory are required for eight-bit samples and 16-bit samples respectively.

The second method of reducing computational load relies on bitwise operations on the IF samples. Called *Vector Processing*, also introduced by (Ledvina et al 2003) this method operates by storing data bits from separate samples in a single word.

2.3 Processor Technologies

The starting point of this work, GSNRx^{TM}, is designed to run on an x86 based, Windows PC. This has the advantage that it can run on most consumer computers. The downside however, is that neither the operating system nor the processor are optimized for real-time processing of highly mathematically intensive tasks such as the high-rate operations described in the previous section. This section presents some available processor technologies and explains why an NVIDIA GPU was chosen as the implementation architecture for this work.

2.3.1 General Purpose Processors

General purpose processors (GPPs), such as the x86 based processor GSNRx™ runs on, are built on the von Neumann architecture (Hennessey & Patterson 2007). As such, they have a single storage structure – memory space – to hold both instructions and data. This means that programs and data can be stored in the same space and loaded on the fly. This has the benefit of simplicity in both hardware and operating system design, enabling higher clock speeds and leaving more resources free for other tasks. The drawback is that the processor can either be reading an instruction or reading / writing data, but cannot do both simultaneously.

This limitation leads to the von Neumann bottleneck, the limited throughput between the processor and external memory. Since in most processors, throughput is much smaller than the rate at which the processor can work, speed is limited when few operations are required on large amounts of data, such as for DRC.

Mechanisms, such as cache and branch prediction, exist in order to mitigate this bottleneck (Jouppi 2007).

Cache is high-speed, low access cost storage for a duplicate copy of data, stored elsewhere, which is more costly to access. In terms of processors, cache is located on the same chip as the processor and duplicates external memory, either in RAM or on the hard

disk. For example, in a modern PC, the cache can be accessed in one to two clock cycles, while RAM access requires 20 clock cycles. A cache controller will automatically fetch commonly used external data. Once it is on chip it can be accessed and changed in the future without need to re-fetch. External access is only needed once it is replaced in the cache with other data.

Branch prediction is a mechanism which attempts to guess which way a branching instruction – if/else, or loop – will go before it is known. Guesses are made based on previous executions of the branch. The guessed branch is fetched and speculatively executed, and placed into the pipeline before the result of the branch is known. If it guessed correctly, a branch predictor ensures that no cycles are lost due to a branch. An incorrect guess is usually quite costly as the entire pipeline – up to 20 cycles – has to be flushed and the alternate branch fetched.

These features, along with high clock speeds, mean that GPPs are well suited to executing low-data, non sequential control code. Since so many resources are devoted to memory management the underlying memory and processor architecture is less important to the programmer and can be treated as transparent.

2.3.2 Digital Signal Processors

In contrast to GPPs, digital signal processors (DSPs) are usually based on the Harvard architecture, meaning they have a separate memory space / bus for instructions and data (Hennessey & Patterson 2007). This means that instructions and data do not necessarily have to be the same size, and more importantly, can be fetched simultaneously.

Common features of DSPs include very complex direct memory access (DMA) controllers, fast highly parallel multiply-accumulate capable logic units, floating point units integrated into the datapath and a highly pipelined architecture. These features mean that DPSs are well suited to high-data, high mathematical complexity, low branching operations needed for signal processing. DSPs also have a highly irregular nature in terms of architecture and instruction sets, meaning that the programmer often has to hand-optimize programs in assembly. The lack of resources devoted to memory management – i.e. cache – means that the underlying structure cannot be treated as transparent and must be taken into account.

2.3.3 Graphic Processor Units

Graphics processing units (GPUs) are specialized, floating point, processors capable of highly parallel arithmetic operations. Originally GPUs contained fixed-function pipelines and were optimized for polygon fill rates (Owens et al 2007). By 2002, however, hardware had become so advanced that even low-end consumer products were capable of refreshing an entire screen up to 1500 times per second. This encouraged manufacturers

to produce flexible hardware and application programming interfaces (APIs) capable of not only rendering but also programmable shading, lens simulation and even real-time ray-tracing (McCool 2001). While earlier hardware was limited to outputting a 8-bit-per-channel colour value, modern GPUs have programmable processing units capable of full IEEE single precision floating point operations (Owens et al 2007).

Over time, this change in hardware produced research interest in general purpose computing on graphics processing units (GPGPU). This has persuaded manufacturers to develop and release even more flexible APIs which allow the programmer direct access to the hardware.

Modern GPUs have a highly parallel architecture with upwards of 128 execution cores, each housing upwards of 32 arithmetic logic units (ALUs). Very few resources are devoted to out-of-order execution, branch prediction or cache, yielding very high parallel processing performance at the cost of code complexity. The hardware cannot be transparent to the programmer. They are designed for high-end graphics, meaning GPUs can best handle many parallel but simple tasks.

As of the outset of the project, there were no standards in either GPU code or hardware structure. The two main manufacturers, NVIDIA and ATI have their own proprietary programming languages and device structures. In the initial phases of this project, it was determined that the NVIDIA development kit was more mature and stable and was thus

chosen as the platform. This decision was based on the available documentation and support from both manufacturers.

During this work, OpenCL – a coding framework which allows compilation and execution on both ATI and NVIDIA GPUs – was adopted by both manufacturers.

A discussion on the NVIDIA programming language – CUDA – and hardware device structure follows.

2.3.3.1 CUDA

Compute Unified Device Architecture (CUDA) is NVIDIA's parallel programming model and software environment built by extending the capabilities of the C programming language. It is based on three main abstractions: a hierarchy of threads, a hierarchy of memories and a system of blocking synchronization.

Using CUDA, a programmer defines C functions – kernels – which are executed *k* times in parallel by *k* threads – see Figure 4. This contrasts with standard C code which is executed once. Each thread is assigned a unique ID which is accessible from within the kernel. Threads are arranged into one, two or three dimensional blocks. Each thread block is subsequently arranged into a one or two dimensional grid. The large number of threads,

and the low context switching / synchronization overhead mean that massive parallelization, to the point of one thread per data sample, is efficient.

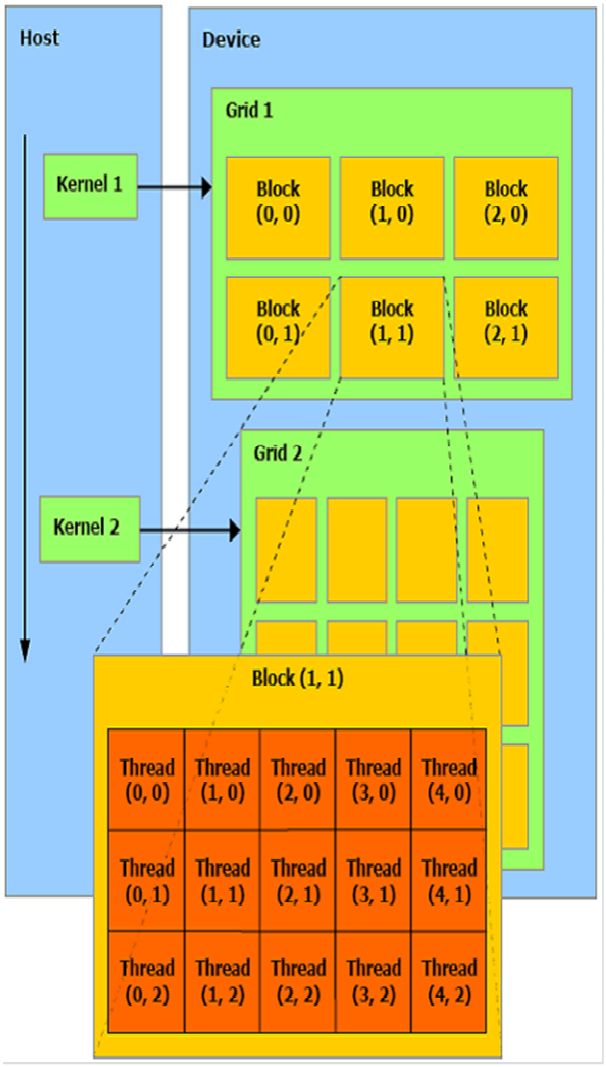

Figure 4 -- GPU Thread Structure (NVIDIA 2009)

2.3.3.2 NVIDIA GPU Structure

In 2006, NVIDIA introduced the Tesla unified graphics and computing architecture (NVIDIA 2009). The scalable nature of this design allows the architecture to span a wide array of segments from high-performance to entry level. The massively multithreaded processor is a highly efficient platform for both graphics and general purpose, parallel computing.

The Tesla architecture is based on an array of multithreaded streaming multiprocessors. Each multiprocessor consists of eight scalar processors, each with its own register memory, two transcendental special function units, a multithreaded instruction unit, an on-chip parallel data cache of shared memory, access to cached read-only constant memory and cached texture memory – see Figure 5.

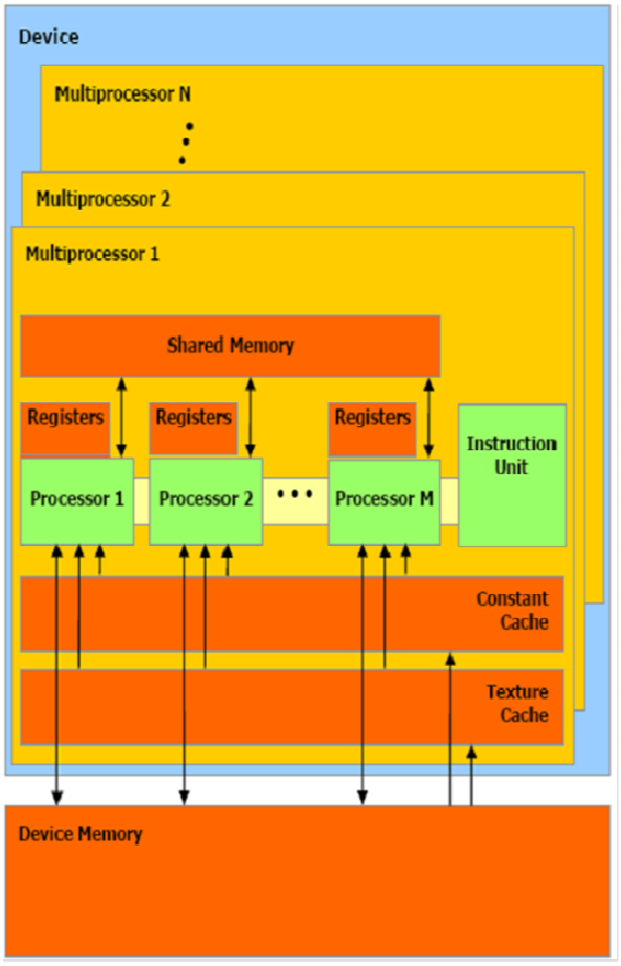

Figure 5 -- GPU Hardware Structure (NVIDIA 2009)

When the host CPU invokes a kernel grid, threads and blocks are enumerated and distributed to multiprocessors with available execution resources. Threads within one block will execute on one multiprocessor, giving them access to low-latency, shared memory.

Hundreds of threads running several different kernels are managed through a Single Instruction Multiple Thread (SIMT) architecture. The microprocessor's SIMT controller creates and schedules groups of up to 32 threads, called warps. On instruction issue time, the SIMT controller selects a warp that is ready to execute and issues the next instruction to all threads simultaneously. If threads diverge due to a data dependant branch – loop, if statement, switch – execution has to be serialized. The controller will execute each branch path, deactivating threads which are not on that path. Separate warps can execute separate instructions at the same time. Thus, to maintain efficiency, it is important to ensure that there is minimal warp-level divergence.

2.3.4 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are reconfigurable logic devices that are designed to implement a variety of digital circuits. They are integrated circuits that contain many identical logic cells – basic logic elements (BLEs) – connected with wires and programmable switches. Each BLE can take on a limited set of functions. Designs are implemented by programming which logic function each cell will perform, and selectively closing switches to connect the cells.

Logic cell architecture varies between different families of FPGAs, but the most common BLE is implemented as a lookup table (LUT) combined with a flip-flop (FF), as shown in

Figure 6. Input size varies from three to ten bits and output size is usually limited to two bits.

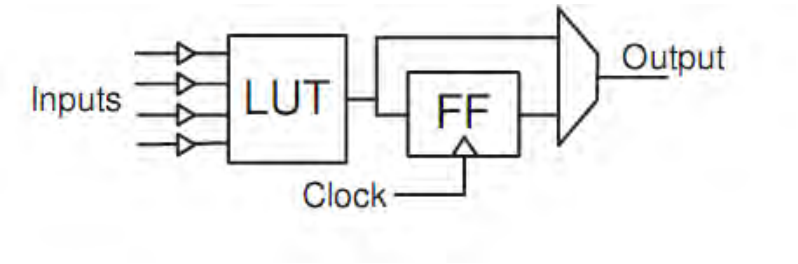

Figure 6 -- Basic Logic Element of an FPGA (Jameison 2007)

Utilization of an FPGA is measured by the number of BLEs required to implement a certain function. It is common for FPGAs to contain additional circuits, such as memory and multipliers (Zhang 2008). In some cases, it is possible to reduce utilization, or implement a wider function by incorporating these circuits.

FPGA design is performed using either a hardware descriptive language (HDL) or schematic capture, with most designs done in HDL as it is more readable for large scale projects.

HDL programming involves creating blocks called entities, enumerating the inputs and outputs of each block, describing the internal logic, and finally connecting blocks. A synthesizer will then translate this code into LUTs and wiring.

In order to expedite designs, vendors supply designers with IP cores – large blocks implementing DSP algorithms, communication modules, interface modules and soft processors (Zhang 2008).

2.3.5 Reasons for Choosing GPU

A decision was made to design a real-time DRC for NVIDIA GPUs. This was based on several reasons: hardware capability, hardware cost, hardware availability and development software maturity.

The fast release cycle of graphics hardware and the highly competitive nature of the market have caused computing power to increase exponentially.

Figure 7 contrasts the capability of high end CPUs and GPUs using billions of floating point operations per second (GFLOPS) as a metric. In terms of raw processing power, GPUs are the better choice.

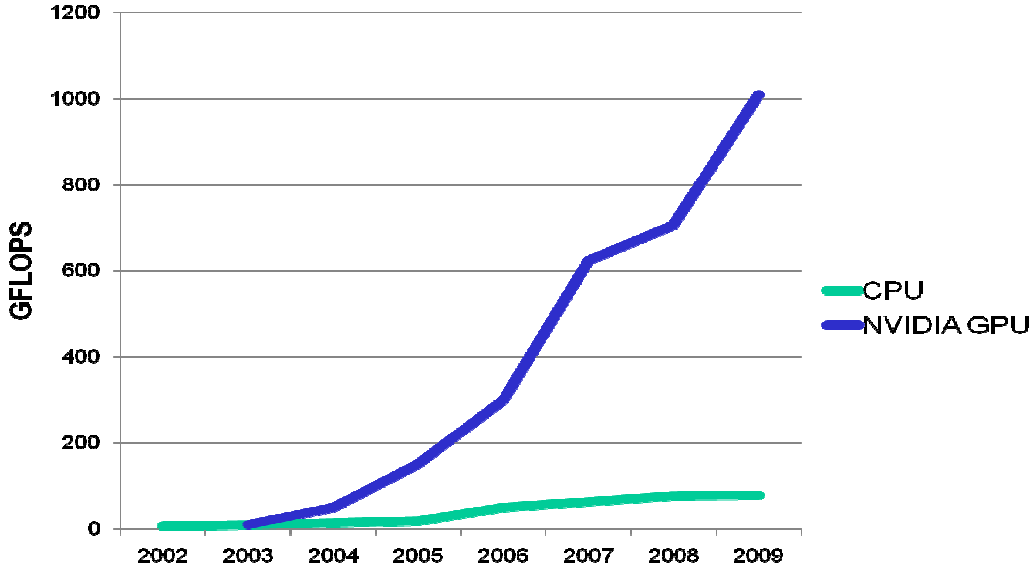

Figure 7 -- Processing power of GPUs compared to GPPs

The increase in semiconductor capability for both GPPs and GPUs is driven by the same advances in fabrication technology. The disparity in performance can be attributed to a fundamental difference in architecture, as discussed in the previous sections. GPPs devote many resources to branch prediction, out-of-order execution and cache while GPUs devote all possible transistors to arithmetic operations. This leads to a device capable of orders of magnitude more operations per second using the same transistor count.

FPGAs are more capable than GPUs for this kind of application as the hardware itself is customizable. A design was started for an FPGA which included a multi-channel DRC module. In spite of this, it was decided to abandon this stream of work for two reasons. First, the hardware is expensive and not readily available in consumer PCs, unlike GPUs.

Each unit would have to be custom equipped. Second, as this work is to be a research platform, it is important that it can be easily maintained. While most researchers are fluent in C and will be able to modify CUDA code with little training, hardware design knowledge and HDL experience is less common.

Finally, the choice between NVIDIA and ATI was made due to development software maturity. Work was started on an ATI system due to the availability of machines within the research group, but was later abandoned due to lack of support and documentation from ATI. NVIDIA's development kit is better documented, with clear examples.

2.4 Algorithm Parallelizability

GFLOPS were discussed in the previous section as a performance metric. This number, however, can be misleading. In order to reach these numbers on a GPU, it is necessary to utilize all of its resources. To do this, the algorithm to be performed must be parallelizable. That is, it must be possible to dissect the algorithm into individual slices which have little or no dependency on each other. These pieces can then be mapped onto multiple independent processors and simultaneously executed.

For certain algorithms, like Newtown's method, parallelization is not suitable. Each subsequent calculation relies on the result of all proceeding calculations, and thus even if

separated into different processors, each calculation would need to wait for all previous calculations to finish.

Some algorithms, such as a brute force search in cryptography, are inherently parallelizable. Each processor can be given the task of checking one solution, and the problem ends when the correct solution is found.

The DRC algorithm lies in between the above examples, requiring both parallel and sequential operations. While removing Doppler and correlating each sample for each channel at the same time is a parallel operation, the results then need to be reduced (i.e. accumulated), which is a sequential operation.

2.5 Conclusion

This chapter presented an overview of GNSS receivers, concentrating on the DRC algorithm and software based receivers. Processing power and memory requirements were quantified and the motivation for using a co-processor was given. Possible co-processor technologies – GPP, GPU and FPGA – were described and a discussion on the parallelizability of the DRC algorithm was presented.

Due to the parallelizability of the DRC algorithm, and the high availability of hardware, a GPU was selected as the co-processor.

The following chapter will describe the development of the DRC module.

Chapter Three: DRC Development

This chapter details the work done to extend the capability of GSNRx™ to facilitate high-speed processing of high-bandwidth signals. It begins by describing abandoned design streams. This includes the DRC design done on the FPGA and on the ATI GPU. Next, the design cycle for the DRC on the NVIDIA GPU is described. This is followed by design challenges for the same.

3.1 FPGA Development

Initially, it was decided to use an FPGA in a co-processor environment to offload the high-rate operations, namely Doppler removal and correlation. A Xilinx Virtex 4 SX35 was chosen as the development platform. The SX53 has 34,560 BLEs, a maximum of 384 kB of distributed RAM and a maximum of 5 MB of block RAM. This was deemed sufficient for the required task.

In the subsequent sections the DRC module is described, and the current status reported and future work needed to complete the project outlined.

3.1.1 FPGA Design Overview

The FPGA DRC module can be broken down into three main parts: an input First-In, First-Out (FIFO) buffer, multiple single channel DRCs and an output FIFO, as shown in Figure 8.

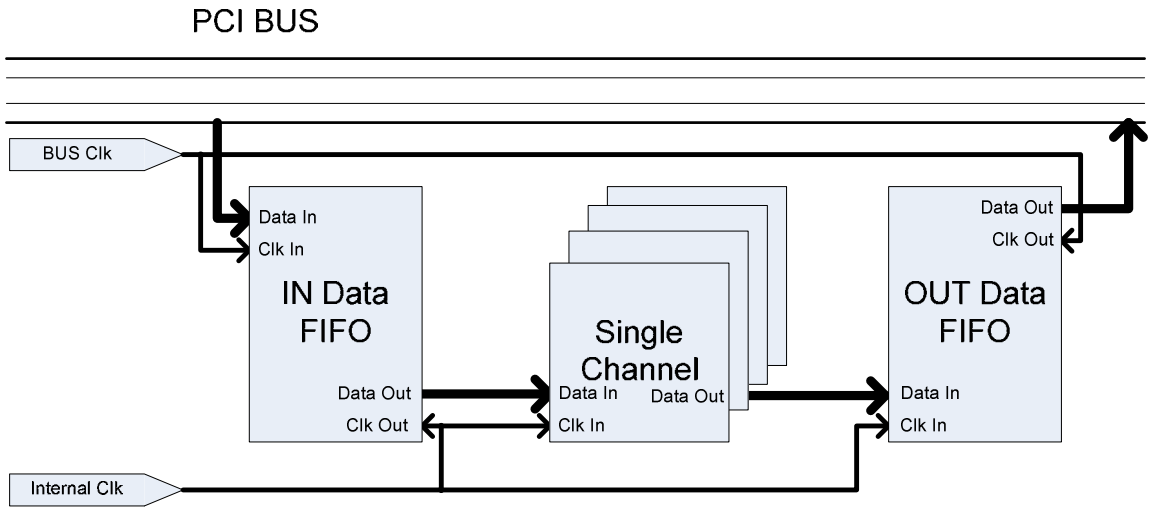

Figure 8 -- FPGA Design Overview

The input and output FIFOs are instantiated IP cores provided by Xilinx. They each consist of a 32x512 block RAM module, and logic to allow the input and output to be clocked by different clocks. Data is fed into the INPUT buffer from the PCI bus, and clocked by the same. After the first word – the size of one sample of data, eight bits in this case – is loaded, a FIFO_EMPTY flag, internal to the FIFO, is cleared, and reading is enabled. Since the only time that reading and writing is performed on the same word is

when the buffer is empty, this method ensures that there will be at least one write clock cycle delay, and that clock domain boundaries can safely be crossed. That is to say, under no condition will the word be available to the read circuitry for less than an *entire* read clock cycle.

The single channel DRC is shown in Figure 9. For simplicity, the internal clock routing, which goes to every multiplier and adder, has been omitted. Note that since clock domains have been crossed in the FIFOs, the entire design can run on one clock.

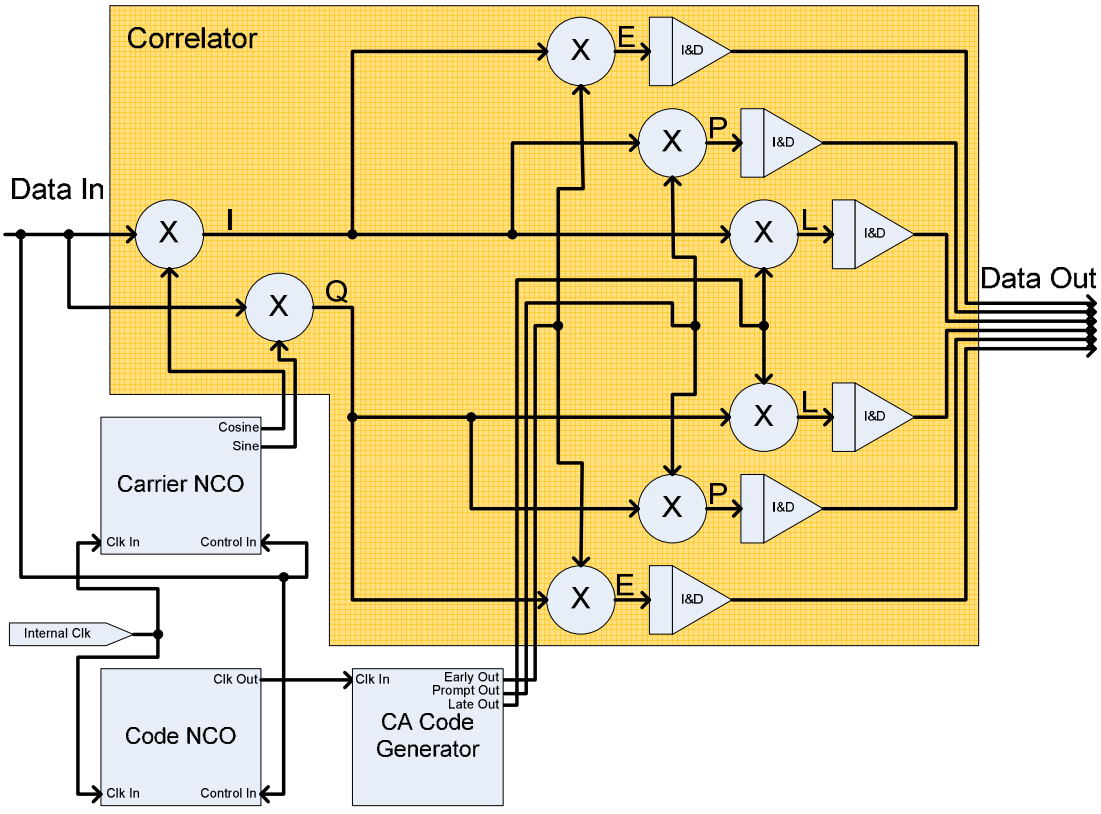

Figure 9 -- FPGA Single Channel DRC

Data is fed into the channels though a three bit wide bus. Special control sequences in the data are used to update the code and carrier numerically controlled oscillator (NCO) frequencies. To expedite design, the carrier and code NCOs are instantiated Xilinx IP cores, like the FIFOs. The carrier NCO generates a four-bit wide sine and cosine signal which is fed into the correlator unit. The code NCO generates a one bit clock which is used to drive the C/A code generator. The C/A code generator relies on two 10 bit shift registers to generate an early code replica. Delay states are used to generate prompt and late.

The first stage of multiplication – carrier wipeoff – is done using a six bit multiplier. While not necessary, the wider output allows for better optimization of the subsequent steps. The second stage of multiplication is performed using eight bit multipliers. Finally, results are summed using 20 bit accumulators.

On a code rollover, data is fed from the accumulators into the output FIFO. Since channels are not synchronous, that is to say their code rollovers happen at different times, and they will not write to the buffer at the same time. In order to ensure no data collisions, results are buffered locally. The WRITE_ENABLE pin on the output buffer is polled before writing, and results are held until it is available.

3.1.2 FPGA Design State

In the current state, the FPGA DRC module is capable of single channel operation. The logic required for the control sequences on the data has not yet been implemented. To complete the module, the control logic would need to be implemented, the design extended to multi-channel and GSNRx™ modified to incorporate this module.

3.2 ATI GPU Development

Due to the availability of ATI hardware within the research group, it was initially decided to develop the GPU DRC module for the ATI Mobility Radeon HD 2600. Unfortunately, even though the Mobility Radeon HD 2600 incorporates the same chipset as the standard Radeon HD 2600, Mobility hardware was not supported by the ATI drivers. To overcome this problem, software was used which changed the device model number stored on the hardware, tricking the operating system into accepting standard drivers for it.

A standalone single channel DRC module was then implemented and tested for functionality and speed. The module was capable of streaming data onto the GPU, processing and streaming results back to the CPU. The module had correct functionality, but showed instability issues when run at a high rate. It is tempting to speculate that this

was due to overheating. The operating system recognized the hardware as standard, not mobile which is capable of running at a higher speed; the clock rate was increased, causing the device to overheat, thus halting the device under full load.

At the time, there was no information from ATI regarding support for mobility hardware. To minimize risk, it was decided to abandon this stream of development and move onto NVIDIA hardware.

3.3 NVIDIA GPU Development

An NVIDIA GeForce 8800 GTX fitted into a PC with a single-core 3.0 GHz processor with hyper-threading technology was chosen as the new development platform. To facilitate finer grained functionality testing of the DRC module, it was developed in sections. The following section describes the development cycle. Following that, challenges faced during development are discussed. Finally, an overview of the DRC module is given.

As mentioned in the introductory chapter, the starting point of this work was a GPU solution developed by the PLAN group (Petovello et al 2008). While this work was used as a reference, a complete redesign and restructuring of the solution was required in order to facilitate higher speed operation.

The work discussed in the remainder of this chapter is based on (Knezevic et al 2010). The developed module is very flexible, and is capable of correlation for any spreading sequence, sampling rate, Doppler frequency and early-prompt-late chip spacing, simply by receiving an initialization parameter from the PC. All GPU code has been developed in CUDA – a modified version of C – and all CPU code has been developed in C++.

3.3.1 Development Cycle

The multiplier, the basic building block of the DRC, was developed first. This kernel was capable of streaming two sets of data from the CPU onto the GPU, multiplying them and returning the results to the CPU. Having this unit allowed to test the transfer bandwidth between the CPU and the GPU.

Next, this multiplier was extended to a single channel, single lag DRC meaning it could multiply an incoming stream of IF samples by a single carrier and code replica. Sine and cosine values were calculated on the GPU per sample and the spreading sequence was stored in a compressed form in constant memory, exploiting the cache available on each multi-processor – see section 3.3.3.1. At this point, functional and preliminary speed testing was performed. As the device could perform correlation of 40 Msamples in under a second, it was determined that the device was fast enough to permit real-time operation and development was continued.

At this point, each thread, or each instance of the kernel running on the GPU, of the multiplication kernel was responsible for the Doppler removal and correlation of a single sample, and there was no capability to sum the results. A reduction kernel was developed next. This kernel was responsible for looping over the results generated by the multiplication module and summing them.

Finally, the DRC kernel was extended to allow multi-channel, multi-lag operation. After functionality and speed testing, the kernel was integrated into GSNRx™.

3.3.2 Development Challenges

Some of the challenges in developing the GPU DRC module are detailed in the following sub-sections.

3.3.2.1 Expensive Copying

Copying data onto the GPU board is expensive. It takes eight GPU clock cycles per word to copy from the CPU or the global device memory. More importantly, there is a 400 to 600 clock cycle latency for memory transfers.

Unfortunately, it is difficult to overcome this problem. The threading engine can hide some of this latency by allowing other threads to run while some are waiting on data, but

the device is not fully utilized at this time. The only thing which can be done to mitigate this problem is to minimize the amount of copying to be done. Even though many threads require access to the same data at different times, it is important to ensure that it is requested only once, and then stored locally in shared memory on the multiprocessor. In order to not overwhelm the relatively small amount of shared memory, it is necessary to ensure that all threads running on one multiprocessor are processing nearby time-slices of the data stream.

3.3.2.2 Large Thread-Count Required

As mentioned in the previous section, in order to hide data fetching latency, the threading engine switches to threads which are not pending on data transfers. In order to do this effectively a large number of threads are required. Since thread count is dependent on sample count per kernel call, thread count is maximized when the time slice of data to be processed per kernel call is maximized. That is, each kernel call should process an entire code period for each channel.

The existing DRC and tracking loop update algorithms in GSNRx™ did not allow this approach. As can be seen in Figure 10, processing was interrupted not only by code rollovers, but by measurement epochs as well. At a measurement epoch, the NCO values were recorded and used to generate measurements for the navigation solution. This stop in processing was also used to synchronize the channel clocks.

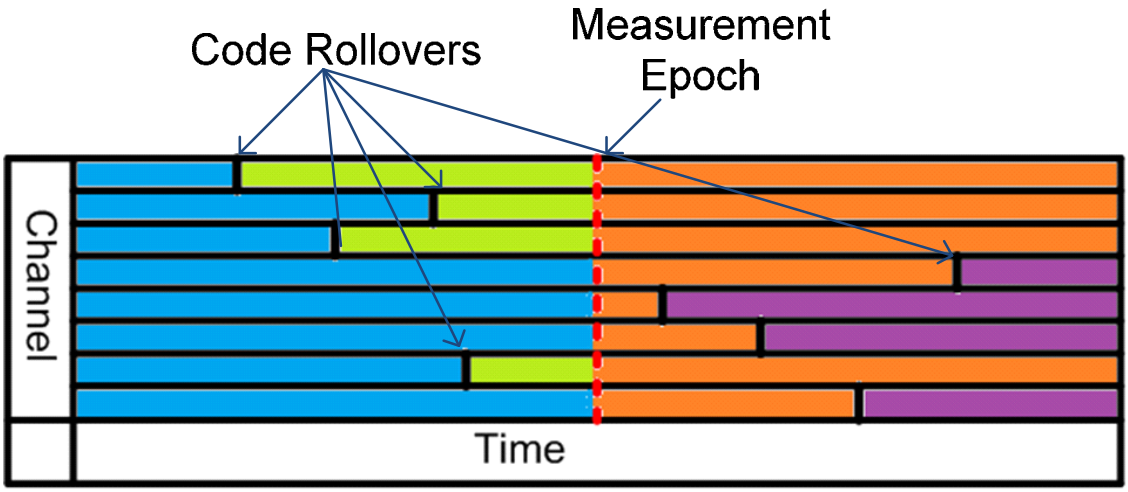

Figure 10 – Synchronous DRC / Tracking Loop Update Algorithm

In order to remove this interruption, an algorithm was developed to allow channels to run independently, as shown in Figure 11. Measurements on each channel are made on code-rollover epochs, and tracking loops and tracking parameters are then updated. In order to provide the simultaneous measurements the navigation solution requires, measurements are propagated either forward or back by keeping track of their rate.

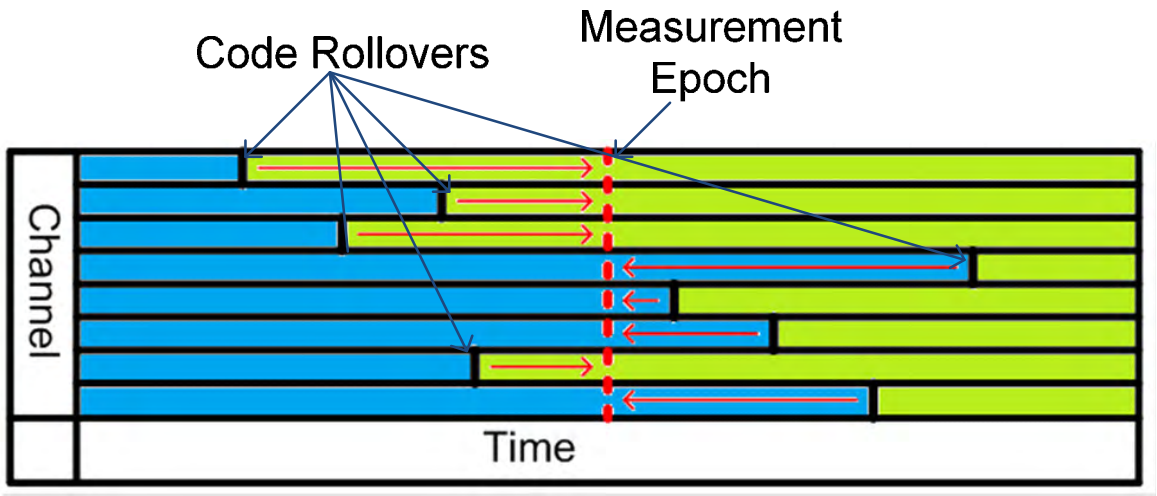

Figure 11 – Asynchronous DRC / Tracking Loop Update Algorithm

The asynchronous algorithm adds complexity to the channels as each has to keep its own clock dependant on the number of samples processed. By keeping at least two code rollovers of data on the GPU at a time in a ping-pong buffer, this method allows each channel to run uninterrupted for an entire epoch.

3.3.2.3 Reduction is Serial

The next challenge faced in designing a parallel DRC module is that reduction, or the accumulation of the results, is an inherently serial operation due to the data dependencies between subsequent steps.

As reduction is a very common operation in parallel processing, many approaches have been developed (Harris 2007, Roger et al 2007, Horn 2005). All are based on the simplest and most intuitive approach, which mimics merge sort. That is, each thread is responsible for adding two elements, and calls are made recursively until all data is reduced. This tree-based approach is illustrated in Figure 12.

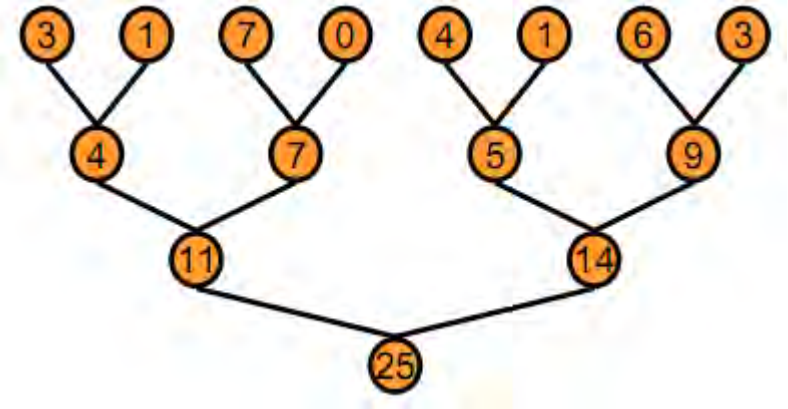

Figure 12 -- Tree-Based Reduction (Harris 2007)

In order to minimize memory latency issues described in Section 3.3.2.1, data is first copied into shared memory, and then operated on. Selecting which two elements each thread will operate on is a non-trivial problem. If, for instance, consecutive elements are selected as shown in Figure 13, loads by consecutive threads will cause bank conflicts, as described in the following paragraph.

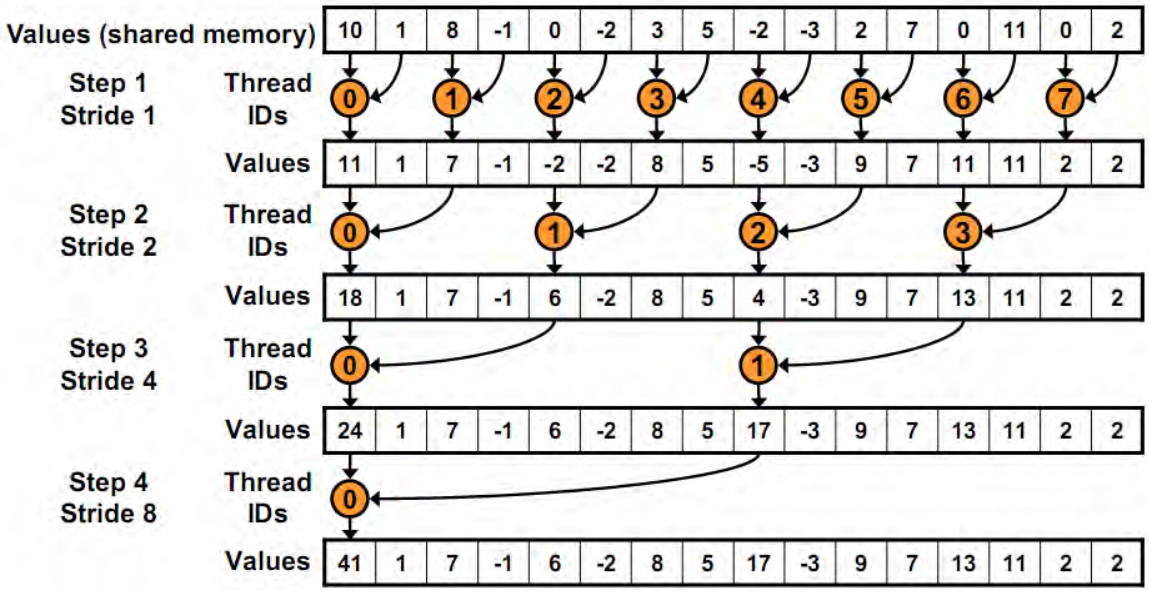

Figure 13 -- Reduction, Sequential Addressing (Harris 2007)

To maximize memory bandwidth, multi-processor shared memory is divided into equally-sized modules, called banks, which can be accessed simultaneously (NVIDA 2009). That is, if *n* threads make read or write requests, and their addresses fall into *n* distinct banks, the request can be served in one clock cycle.

If, however, two or more addresses fall in the same bank, a bank conflict occurs and reads are serialized. The hardware logic splits requests into as few reads as possible in order to eliminate bank conflicts. The data stored in shared memory is organized such that each successive word is in a successive bank. Thus, assuming there are eight memory banks and 16 words, in the case shown in Figure 13, threads 0 and 4, 1 and 5, 2 and 6 and 3 and 7 will conflict.

To mitigate this problem, reads have to be addressed as shown in Figure 14. The stride – the number of words each thread should skip – is dependent on the number of banks. In the case of this project, stride is dependent on the number of threads per block. Since there are 16 banks, an optimal value of 16, 32, 48 or 64 provides the highest bandwidth. However, since thread count must also be optimized and has a higher impact on performance, peak processing speed is achieved at a sub-optimal stride of 80. This is further discussed in section 5.2.2.

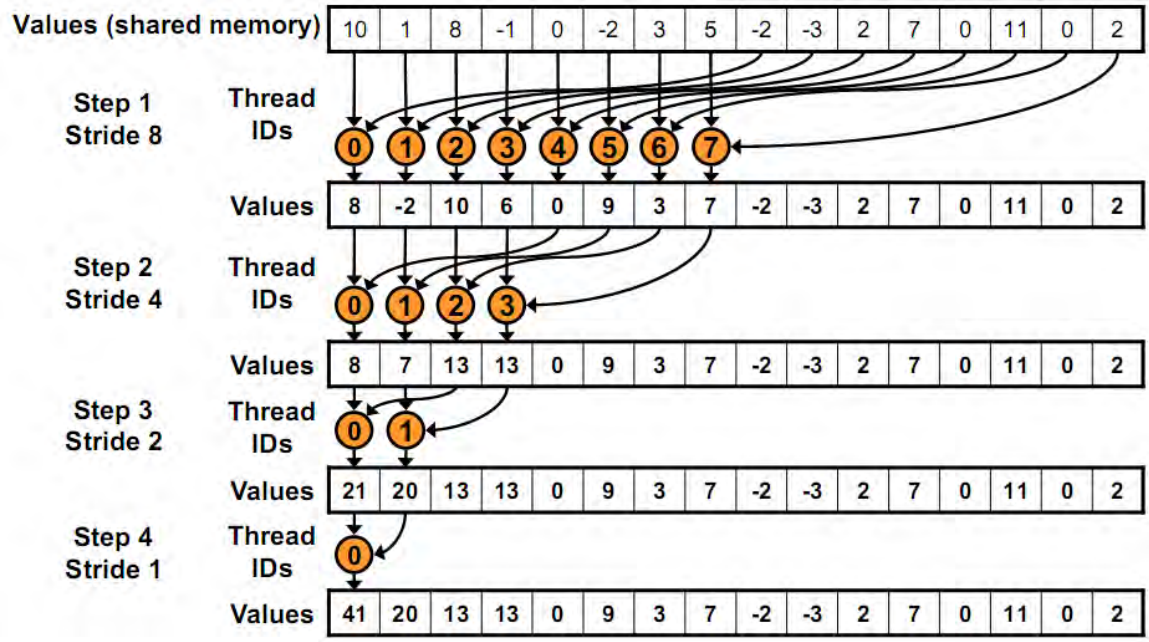

Figure 14 -- Reduction, Optimized Addressing (Harris 2007)

While from a parallel processing perspective, having each thread reduce only two elements is optimal, threading overhead and maximum thread count may make this impractical. Therefore, depending on the number of samples needing reduction, it may be more efficient to have each thread handle multiple samples. This is the case in the developed module. It is done in unrolled loops – loops hardcoded by repeating code, rather than using loop commands – due to loop overhead.

3.3.3 DRC Algorithm

The DRC algorithm can be divided into three main parts, copying data onto the board, the multiplication kernel and the reduction kernel. Each is described in detail in the following sections.

3.3.3.1 Copying Data

As described in Section 3.3.2.1, it is expensive to copy data on to the GPU on-chip shared memory. It is, therefore, important to perform this transfer such that bandwidth is maximized. This section will discuss transfer of IF data onto the board and transfer of the spreading code sequence onto the board.

The transfer of IF data can be broken down into two parts: copying from external RAM onto on-board global memory and copying from on-board global memory to on-chip

shared memory. The former is an asynchronous process which can be performed while the CPU is occupied with other tasks. After an epoch of data has been processed, before the CPU has updated the tracking loops and generated a navigation solution, a transfer is initiated. By the time the CPU is to process more data, the transfer is mostly completed. More timing results will follow in Chapter Five. As such, this transfer is not of great concern.

Transfer from on-board global memory to on-chip shared memory, conversely, is not asynchronous, so optimizing it is important. As discussed in Section 3.3.2.3, shared memory bandwidth is maximized when individual threads are accessing separate banks. Also, even though each thread requires 8-bits of data, the word size is 32-bits. Thus, to maximize bandwidth, before processing begins, a quarter of the available threads are responsible for copying the required data into shared memory while other threads are suspended.

Transfer of the spreading sequence is carried out differently. Since the spreading sequences are relatively small and used many times they can take advantage of the constant cache available on the multi-processors.

Before upload to the GPU, spreading sequences are compressed such that each chip takes up one bit of memory to ensure that the entire sequence will fit into the cacheable memory. The decompression of the sequence requires a modulus operation. While this is

expensive on the GPU, by ensuring that the operation is 2^n, it can be replaced with a bit-wise AND with 2^n - 1.

The following section will discuss the multiplication kernel.

3.3.3.2 Multiplication Kernel

The multiplication kernel is the one capable of benefitting the most from parallelization. Since each sample can be independently operated on, there is no data dependency, and all threads can, in theory, operate simultaneously. In order to be able to cover an entire code rollover epoch – see Section 3.3.2.2 – each thread needs to operate on several samples. Following the stride rules outlined in Section 3.3.2.3, each thread addresses samples to avoid bank conflicts.

First, thread and block indices are used to calculate the *sampleOffset*, which combined with the *startIndex* gives the initial data index as follows:

$$\begin{aligned} sampleOffset &= (b_x \bullet samplesPerBlock) + (t_x \bullet samplesPerThread) \\ dataIndex &= startIndex + sampleOffset \end{aligned} \tag{3.1}$$

where: *startIndex* is the index of the first sample for that channel,
b_x is the block index,
samplesPerBlock is the number of samples processed by each block,
t_x is the thread index,
samplesPerThread is the number of samples each thread must process,

samplesPerThread is calculated as follows

$$samplesPerThread = \frac{samplesPerEpoch}{blocksPerGrid \bullet threadsPerBlock} \tag{3.2}$$

where: *samplesPerEpoch* is the number of samples per 1 ms of data,
blocksPerGrid and *threadsPerBlock* are adjustable design parameters.

blocksPerGrid and *threadsPerBlock* are optimized to ensure that *samplesPerThread* is equal to a multiple of the number of banks on the hardware. This configuration ensures that consecutive threads read data from different banks, thus increasing the bandwidth. Each thread processes *samplesPerThread* samples by looping *dataIndex* from its initial value to *dataIndex*+*samplesPerThread*.

The *sampleOffset* combined with the sample rate, Doppler and initial phase, is used to generate the carrier phase of the sample being operated on as follows:

$$carrierPhase = initialCarrierPhase + sampleOffset \bullet \frac{Doppler}{sampleRate}. \tag{3.3}$$

The code phase is similarly generated. Sine and cosine values are generated on the fly, as opposed to using a look-up-table. This is done because on the fly sine/cosine generation is a very inexpensive operation on the GPU requiring only eight clocks cycles. Memory look-up, however is more expensive due to the delay described in section 3.3.2.1. The required code chip is decompressed as described in the previous section. Since code chip spacing is always kept at or below one-half chip, it is guaranteed that either the early or

late replica will be of the same chip. Thus, two decompressions need to be performed for the three signals.

The integer samples are converted to single precision floating point in order to make use of the floating point ALU. The *–use_fast_math* compiler option is used to allow non IEEE precision floating point calculation.

Figure 15 illustrates how processing is divided amongst threads. To simplify the graphic, sequential addressing is shown. Threads from one block are responsible for a single channel.

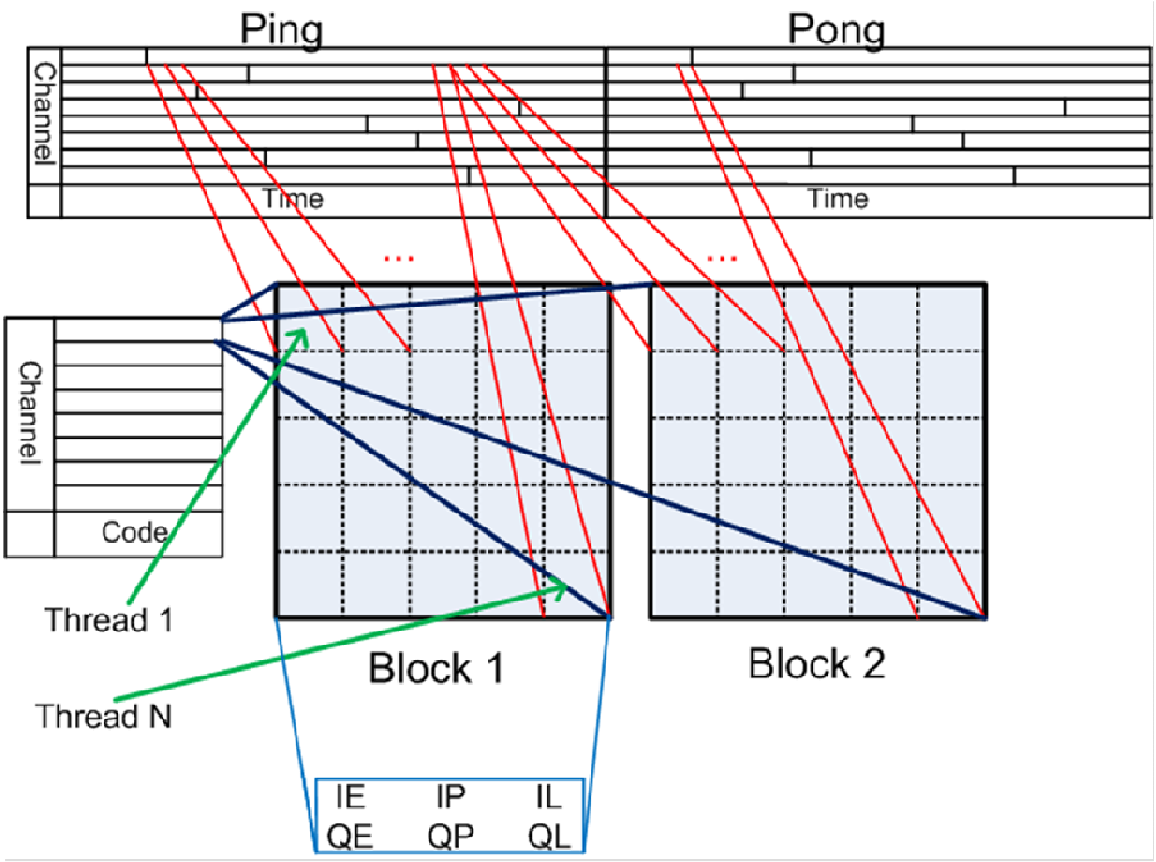

Figure 15 -- Multiplication Kernel

The table on the left hand side of the figure represents the spreading sequences, stored as memory codes on the device. The blue lines, all eminating from one element of the table, represent that multiple threads, and multiple blocks all process data for a single channel. That is, more than one block can be used to process a single channel. The six results from each thread – early, prompt and late, in-phase and quadrature-phase, represented as IE, IP, IL and QE, QP, QL – are stored in local shared memory. To avoid copying a large amount of data out, one thread per block reduces intermediate results for the entire block.

The following section describes the reduction kernel.

3.3.3.3 Reduction Kernel

The reduction kernel is responsible for adding the results of the multiplication kernel. The stride rules outlined in Section 3.3.2.3 are followed to minimize bank conflicts. Figure 16 visualizes the reduction process. The block on the right hand side of the figure represents a block of threads, each summing the IE, IP, IL and QE, QP, QL results produced by the multiplication kernel.

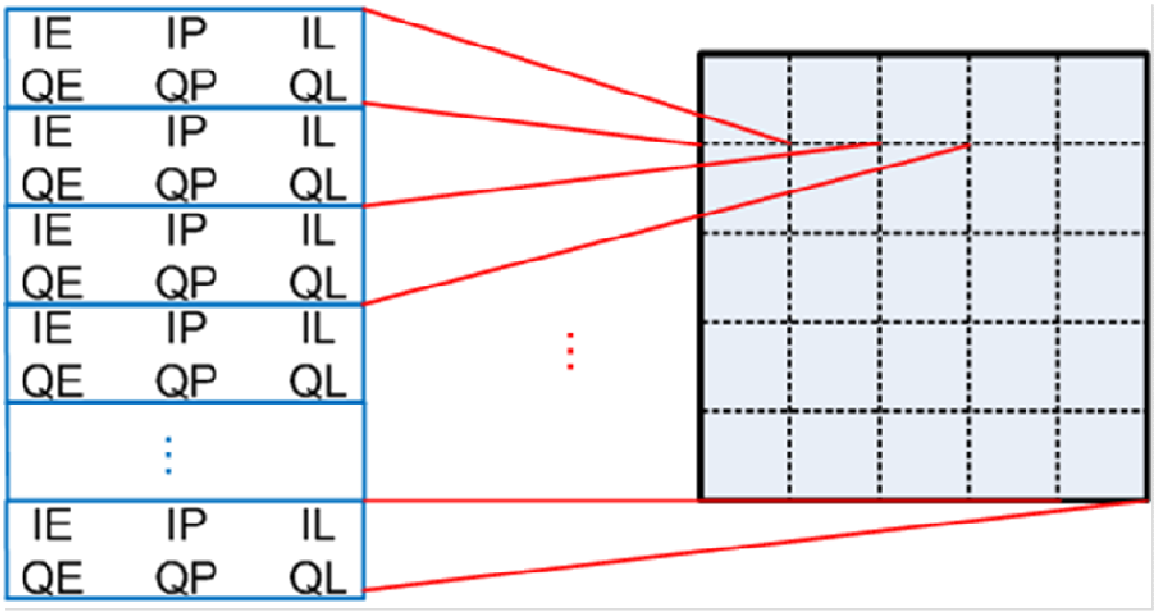

Figure 16 -- Reduction Kernel

As described in Section 3.3.2.3, it is more efficient for each thread to sum over multiple samples. Only one block per channel is used. If more than one block per channel were to be used, multiple kernel calls would need to be made as each block only has access to its own shared memory, thus multiple layers of reduction would need to be performed to obtain a total sum. While this limits hardware utilisation to 50 percent – as there are 16 total multiprocessors and only eight channels – it is faster than performing two iterations due to the overhead of making a kernel call.

3.4 Conclusion

The development process of the high-speed, parallel DRC module was described in this chapter. Specific details required for efficient processing on NVIDIA GPUs are taken into account.

The following chapter will detail the development work done for the real-time sample-source, required for real-time operation.

Chapter Four: Real-Time Operation Development

Faster than real-time processing is only part of the solution for a real-time capable software GNSS receiver. A system is needed which will get samples from the front-end to the PC in real-time, buffer them while the receiver and PC are busy with other tasks and control program flow between receiver tasks and front-end tasks. This chapter will discuss the development of a multithreaded sample-source, an SiGe front-end driver and an Ethernet front-end driver.

4.1 Multithreaded Sample Source

The multithreaded sample source is at the heart of the real-time receiver. As the name suggests, it is responsible for providing the receiver with IF samples as it is ready to process them. In order to do this it has to buffer IF samples if the receiver is not ready to process them, ensure that the front-end is serviced within a specified time such that its buffer does not overflow and block receiver processing if no data is ready. For the purpose of this discussion, the functions of the receiver which process data and the functions which provide data are referred to as the *sink* and *source*, respectively.

In order to be able to perform these tasks asynchronously and to be able to utilize multiple cores, if available on the hardware, the sample source is multi-threaded. The

source – see sections 4.3 and 4.4 – is placed in a high-priority thread which is controlled by interrupts or blocking calls while the sink is in a low-priority thread controlled by the blocking circular buffer – as described in the section 4.2.

4.2 Blocking Circular Buffer

The blocking circular buffer is used to store incoming IF samples and to control program flow.

Buffering of the IF samples is achieved by dividing the available memory space into equally sized blocks, each the size of 1 ms of IF data. The start and end of the buffer are conceptually connected end-to-end as illustrated in Figure 17. This is implemented by ensuring that the buffer size is a power of two and bit-wise ANDing memory increments with the buffer size less one. For instance for a buffer size of 0x10, a pointer at 0xD incremented by 0x4 would yield:

$$\begin{aligned} 0xD + 0x4 &= 0x11 \\ 0x11 \,\&\, (0x10 - 1) &= 0x1 \end{aligned}. \tag{4.1}$$

From the point-of-view of the source, this provides a seemingly infinite buffer to which it can continuously write. As long as the sink can keep up with processing – which the

control segment of the buffer ensures, as described further below – data will not be overwritten.

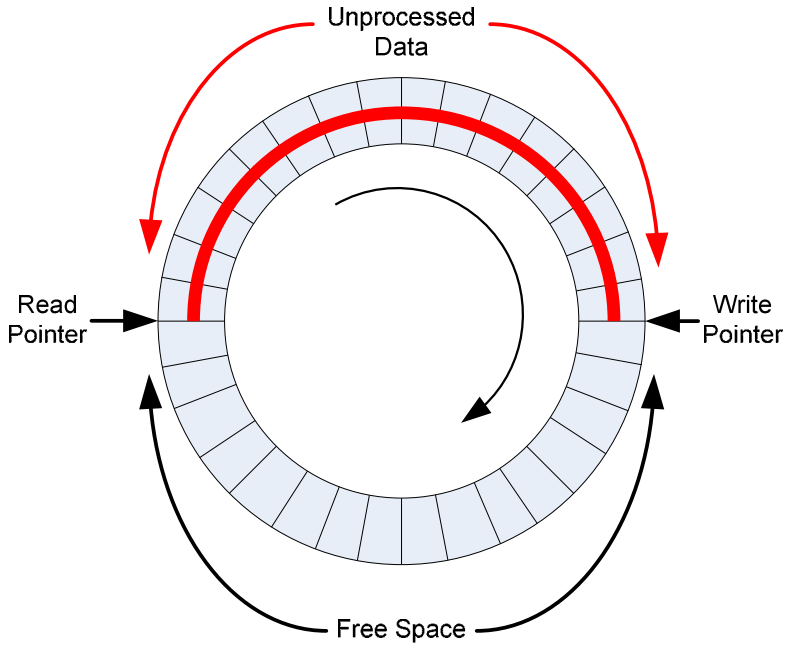

Figure 17 -- Circular Buffer

Program flow is controlled by using the Microsoft Windows multithreading API and semaphores. Semaphores are protected data types which control access to a common resource in a parallel programming environment. Conceptually, a semaphore is a counter of the number of free resources available. When a resource is used, the semaphore is decremented and when a resource is released, the semaphore is incremented. When a request for a resource is made and the semaphore is already at zero, it will block the caller until a resource is available. When a buffer is instantiated, read and write pointers

and read and write semaphores are created. The pointers are both initiated to the first element in the buffer. The read semaphore is initiated to zero and the write semaphore to the size of the buffer.

When a buffer write occurs – a source event – the write semaphore is first pended on – ensuring that unprocessed data is not overwritten – and then a read semaphore is released – allowing the sink to utilize this data. Conversely, when a buffer read occurs – a sink event – the read semaphore is pended on – ensuring that there is data to process – and then a write semaphore is released – freeing a spot for new data. If, at a source event, the buffer is full, the source is blocked and data is lost. This condition is fatal to the operation of the receiver. However, as long as the sink is, on average, faster than the source, and the buffer is large enough to accommodate temporary variations in speed, this will not occur. Speed variations can occur due to either operations of the PC external of the receiver or additional receiver processing such as navigation solution estimation; they cause data backlog and filling of the buffer. Once the sink is performing at its normal speed, the buffer is again emptied and processing is again done in real-time.

Since the source and the sink operate independently in separate threads, this set-up allows high-speed servicing of front-end interrupts.

4.3 SiGe Front-End

Initial real-time development was done using the SiGe GN3S front end (University of Colorado at Boulder, 2010) – see Figure 18.

Figure 18 -- SiGe Front-End

The front-end comes in two versions; V1, built around the SiGe SE4110, provides real, two-bit samples at 16.3676 Msps while V2, built around the SiGe SE4120, provides complex I/Q one-bit samples at a frequency of 8.1838 Msps. Both versions have a 64 kB hardware buffer, meaning that they require servicing approximately every 4 ms as per the following:

$$\frac{65536 samples}{16.368 Msamplesps} \approx 4ms\,. \tag{4.2}$$

A driver was developed based on the softGPS Project driver (Danish GPS Center 2010). In order to be able to handle the high-speed servicing requirements, Windows event and timer functions are used to generate interrupts when data is ready from the front-end.

When an interrupt occurs, a look-up-table (LUT) is used to generate signed samples from the one or two bit data as per Table 2.

Table 2 -- GN3S LUT

Incoming Data	Sample (V1)	Sample (V2)
0	1	1
1	-1	-1
2	3	Undefined
3	-3	Undefined

Following sample generation, data is copied into the circular buffer, allowing the sink to run as described in the previous section. A new timer event is loaded so that the following read also generates an interrupt.

4.3.1 Development Challenges

There were two key challenges in modifying the SiGe driver and incorporating it into GSNRx™: overcoming a limitation in the run-time of the device and incorrect data sheet information.

The SiGe front-ends – versions one and two – are firmware limited to 40 s of run-time. That is, there is a counter in the firmware (University of Colorado at Boulder 2010)

which disables the device when it reaches a threshold. Since the firmware is based on GPL modules (Free Software Foundation 2007), the source has been made available. It has been modified to remove the 40 s limitation and to change the device ID so that the OS can recognize it as new hardware. The MinGW GCC Compiler suite (MinGW 2009) and the Small Device C compiler (SDCC 2009) were used to compile the code. Publically available software, called fx2_programmer (Volodya 2002) was used to upload the firmware to the device. In order for fx2 programmer to be able to run, it requires the libusb_win32 driver (ste_meyer 2010).

The procedure for initiating the SiGe front-end is as follows:

1) Mount the device using the libusb-win32 driver
2) Use fx2_programmer to upload the modified firmware to remove the 40 s limitation
3) The device is recognized as new hardware
4) Mount the device using the modified softGPS Project driver
5) Start GSNRx™.

The data sheets provided by the University of Colorado at Boulder contain two errors. First, for version 2, the listed IF is 38.4 kHz while the actual IF is 38.5 kHz. Next, for version 1, the listed IF is 4.092 MHz while the correct IF is 4.1304 MHz and the listed sampling frequency is 16.368 MHz while the correct frequency is 16.3676 MHz.

4.4 Ethernet Front-End

The Ethernet front-end software was developed to allow the processing hardware and the sample provider to be housed on separate PCs. This allows much greater flexibility in processing data. For instance, a file stored on a RAID array can be processed by a separate machine located on the network, or a real-time front-end can be connected to a processing PC. Both of these setups were used in this project – see Chapter Five. In the future, the developed software will allow any front-end acquired by the lab or any PC with access to stored samples to be used as a sample provider, and any PC running GSNRx™ as the receiver.

The Ethernet front-end is divided into two parts which run on two separate machines: the sample provider, which runs on the hardware front-end; and the sample user, which runs on the processing PC. The sample user thread is integrated into the circular buffer in GSNRx™. It uses the Windows sockets service to set up a listening socket to wait for incoming connections. Once a connection is established by the sample provider, the thread enters a loop of blocking calls to read the socket data followed by writes to the circular buffer, as shown in Figure 19. This configuration allows the higher priority source thread to give up control to the sink thread while it waits for data. Unlike the SiGe front-end, the Ethernet front-end is not interrupt controlled.

On start, the sample provider opens a socket to the sample user, and enters a loop of reading samples from the hardware, converting them to eight-bit in order to increase network transfer bandwidth and writing to the socket.

TCP/IP communication is used to ensure that there are no missing samples from the stream by requiring the source to acknowledge the reception of each packet. While UDP would provide faster streaming capability, as no acknowledgement packets need to be sent, a dropped packet would lead to an unknown delay in the stream and the tracking loops would lose lock.

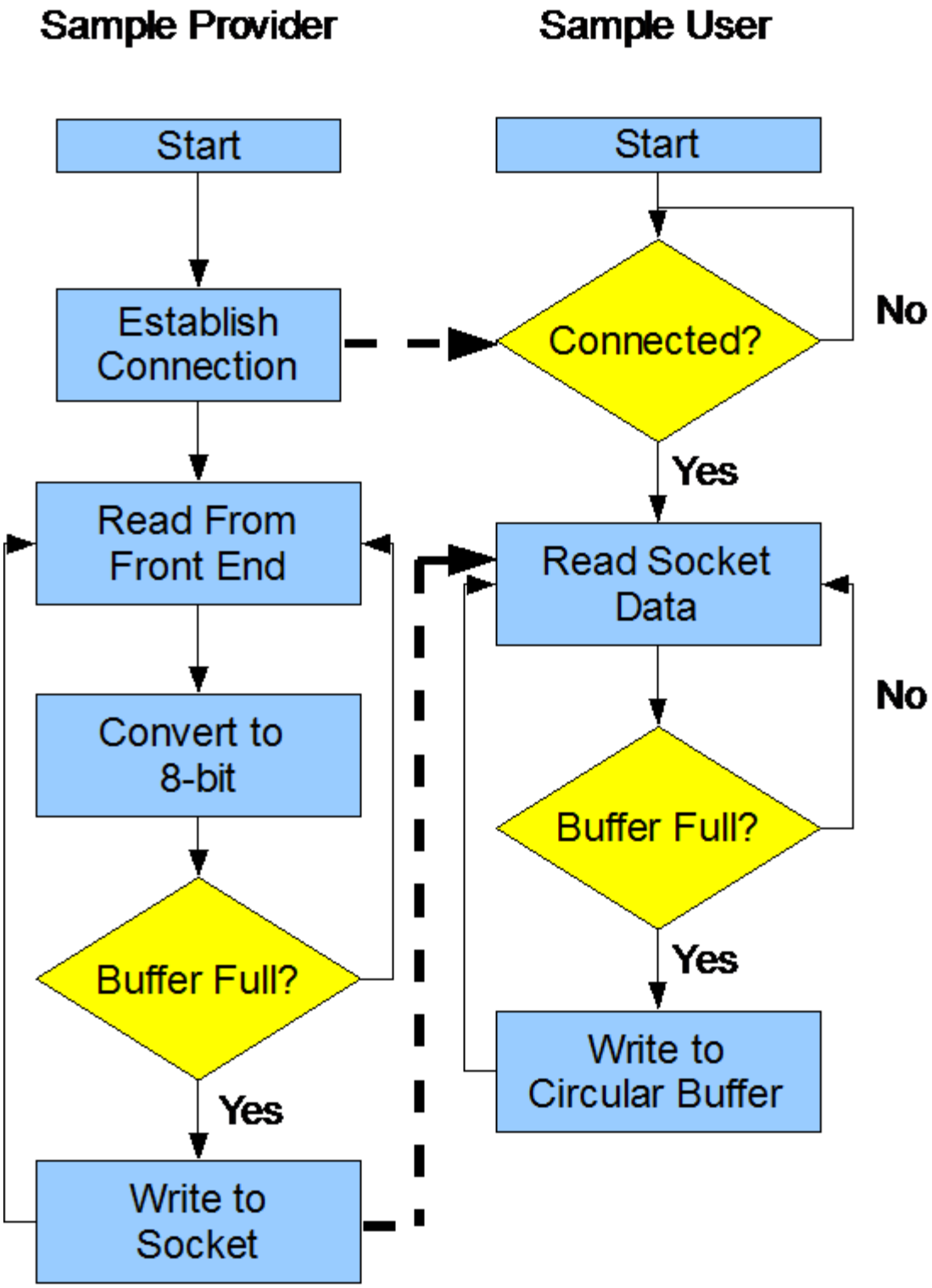
Sample Provider
Sample User
Start
Establish Connection
Read From Front End
Convert to 8-bit
Buffer Full?
Yes
Write to Socket
Start
Connected?
No
Yes
Read Socket Data
Buffer Full?
No
Yes
Write to Circular Buffer

Figure 19 -- Ethernet Sample Source Flowchart

The following chapter will describe the tests performed to ensure correct and stable operation and provide the results of those tests.

Chapter Five: Testing and Results

This chapter is divided into two major subsections: testing methodology and results.

5.1 Testing Methodology

Testing of the receiver was performed in multiple steps. The following subsections detail the profiler structure, the local post-processed test, the Local Area Network (LAN) post-processed test and the LAN real-time test.

5.1.1 Profiler Structure

Initially, an off-the-shelf profiling solution from Intel called VTune was used to profile the performance of the receiver. VTune interrupts the program at either event calls or regular interrupts and records the program counter. It then aggregates this data to show the user what percentage of run-time was spent in each function. High resolution – down to a single line of code – can be obtained with this method.

While this approach performed well on the CPU, code executing on the GPU does not rely on the program counter, and as such does not result in valid timing results. As such, it was decided that the most accurate approach was to develop a custom profiler.

Using the Windows 'QueryPerformanceCounter()' functions (MSDN 2010), a wrapper was built for each function which needed profiling. These functions allow access to the high-resolution performance counter hardware. Resolution is hardware dependant, and was 3.2 GHz on the processing PC. Profiled functions included all kernel calls to be executed on the GPU and a sum of the rest of the receiver processing. Testing showed that the overhead of the timing calls was minimal due to the fact that they were called once every entire code epoch, or 1 ms of data, and not every sample. More importantly, as this overhead was constant over all calls, it did not alter relative performance. Due to the nature of the operating system, the user cannot know when processing will be interrupted to allow other tasks to run. In order to minimize the impact of these interruptions on timing information, and to provide a more accurate estimate of run time, the receiver was run for at least 1000 epochs for each case and data was aggregated.

This type of profiling does not provide direct information on GPU hardware utilization, thus optimization of parameters such as threads per block and blocks per grid is based on minimizing run time.

5.1.2 Local Post-Processed Test

Initial tests were performed by post processing local data. A 120 second 15 Msps IQ sample data file was collected using a National Instruments PXIe-1075 data collection chassis which houses an NI PXI-5690 pre-amplifier, an NI PXI-5600 down-converter and

an NI PXIe-5622 16-bit digitizer – see Figure 20. Only GPS L1 C/A signals were analyzed. At the time of collection, there were 10 visible satellites but only seven were processed – PRNs 2, 4, 7, 13, 16, 20 and 23 – since the receiver is capable of eight channel operation and one channel is used as a noise floor estimator. Samples were recorded at 16-bit precision but converted to eight-bits in order to increase hard-disk transfer bandwidth.

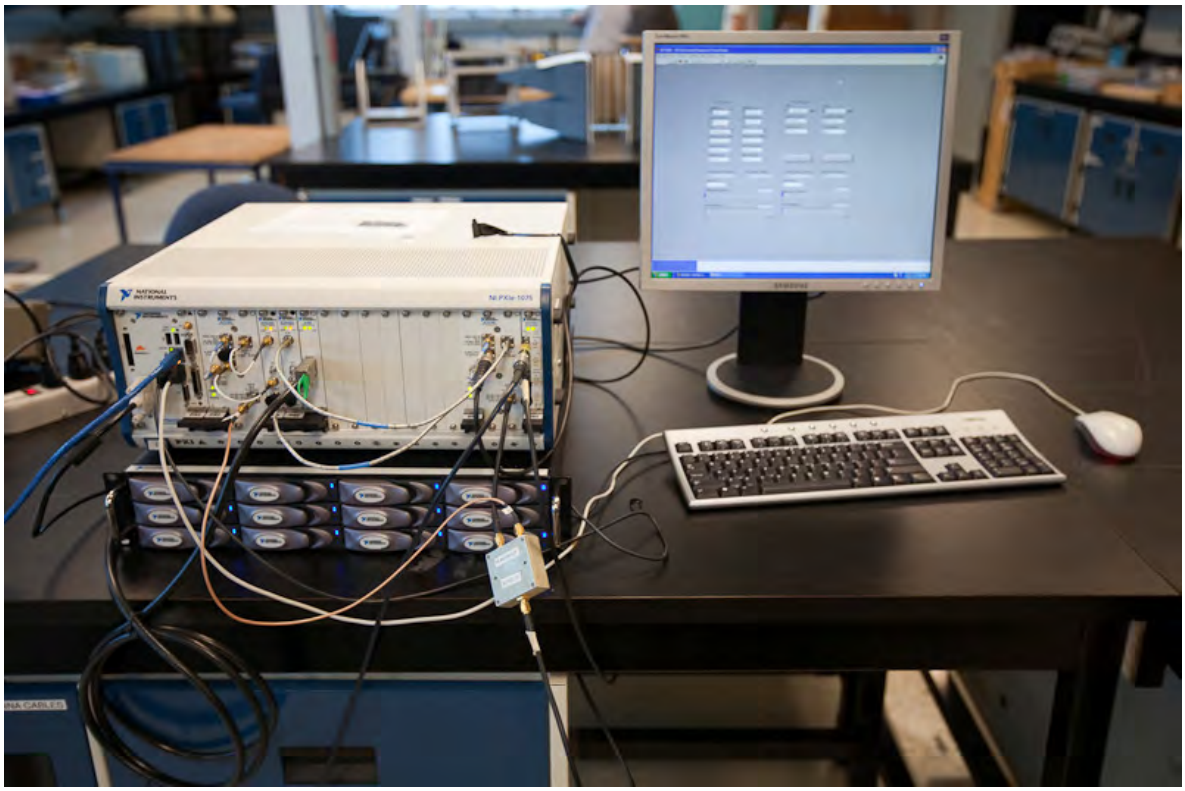

Figure 20 -- NI PXIe-1075

It was soon discovered that disk read and seek times are a bottleneck to processing speed. As it runs, GSNRx™ produces multiple output files including the navigation solution, channel information, tracking information. Tracking and channel information is generated and files are updated every correlation epoch. This limits the speed of processing locally stored data because the hard disk has to constantly seek from the input data file to the multiple output files.

This problem was mitigated by disabling all on correlation epoch, high-speed output information – such as tracking status. A significant increase in speed was observed as seek times were reduced; however the hard disk read speed was still thought to be a limiting factor.

5.1.3 LAN Post-Processed Test

In order to completely remove the hard disk speed as a limiting factor in processing, the Ethernet front-end was introduced. A Gigabit Ethernet card was fitted onto the processing PC and the sample provider was housed on a PC with a RAID 0 array.

A test was performed to evaluate the bandwidth of the hardware. Data was streamed from the RAID array across the network and deleted on the other side. The tests showed that the Ethernet front-end was capable of streaming at 80 MBps. Since the objective of this

work was to develop a receiver capable of 40 Msps operation, the streaming capability is twice as fast as what is required; read speeds were no longer an issue.

5.1.4 LAN Real-Time Test

For high bandwidth, real-time testing, the sample provider runs on a National Instruments data collection chassis. Samples are converted from 16-bit to 8-bit in real-time to maximize usable network bandwidth. The sample user runs on the PC fitted with the NVIDIA card.

5.2 Results

A multitude of tests were performed to characterize the performance of the new DRC module. The following subsections detail functionality testing, timing results and real-time operation.

The nomenclature adopted uses ‘new’ and ‘old’ to refer to the GPU powered DRC and CPU powered DRC, respectively.

5.2.1 Functionality Testing

Initial testing was aimed at confirming correct functionality of the DRC module. Three factors were considered: correlator output, tracking and navigation solution.

Figure 21 shows the overlapped prompt correlator outputs of the new and old version of the DRC module when tracking PRN 23. Small differences are to be expected since the new module performs all calculations using floating-pint arithmetic and floating point sine / cosine values, while the old module uses integer arithmetic and a 3 bit sine / cosine lookup table. These differences can be considered to be numerical approximation errors.

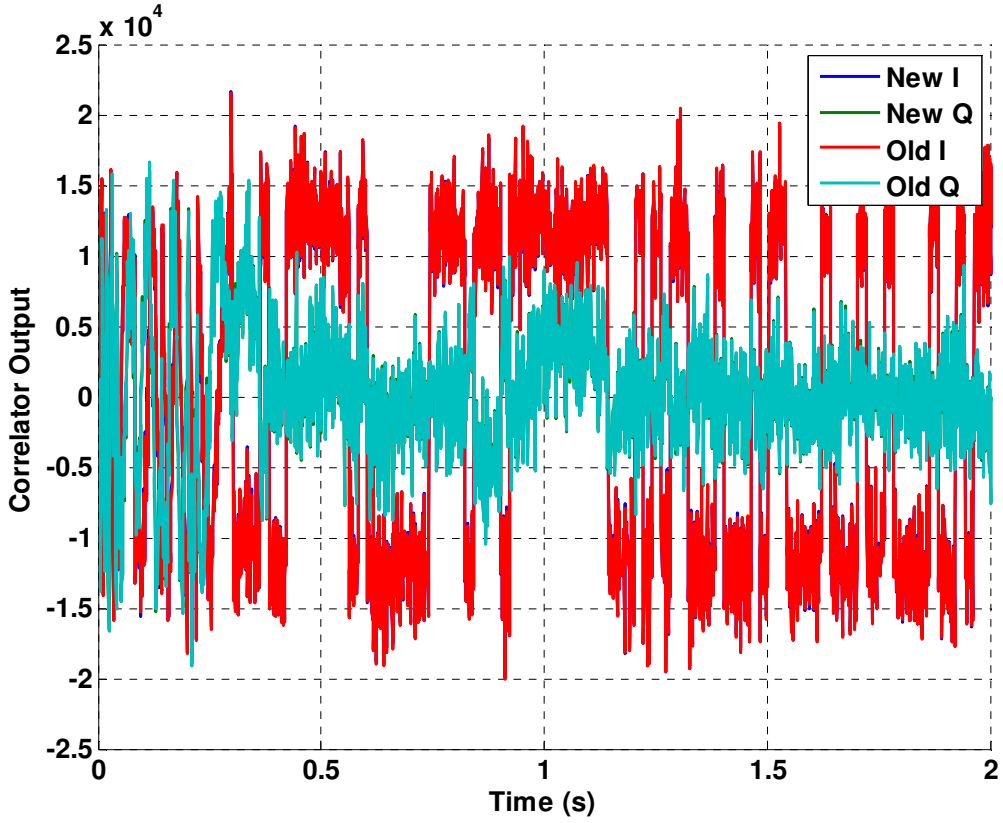

Figure 21 – Prompt Correlator Output when Tracking PRN 23

Figure 22 shows the estimated Doppler frequency of the new and old DRC module when tracking PRN 23.

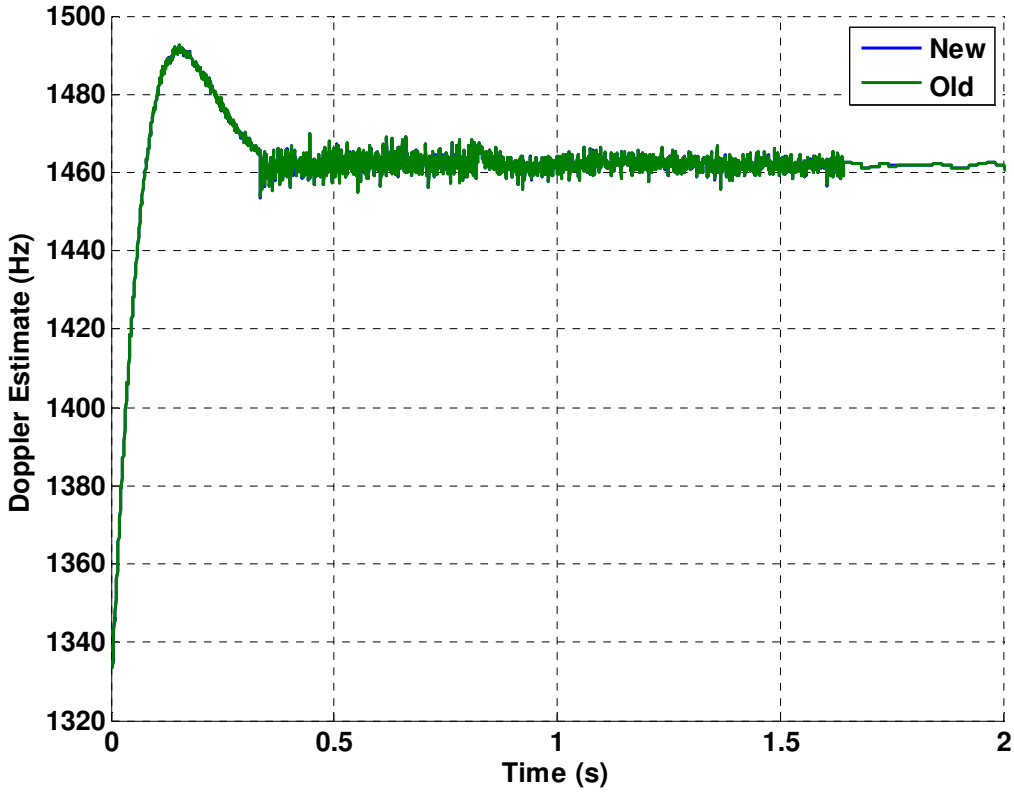

Figure 22 -- Doppler Frequency Estimate for PRN 23 of Old and New Receiver

The transient is very similar in both cases as identical acquisition information is provided and loops have the same parameters. The Doppler estimate becomes much smoother once navigation bit synch is established around 1.7 s and 20 ms correlation is used. Figure 23 shows the difference in the Doppler estimate between the two receivers. It is notable that the difference is zero-mean and under 100 mHz once bitbit synch is established.

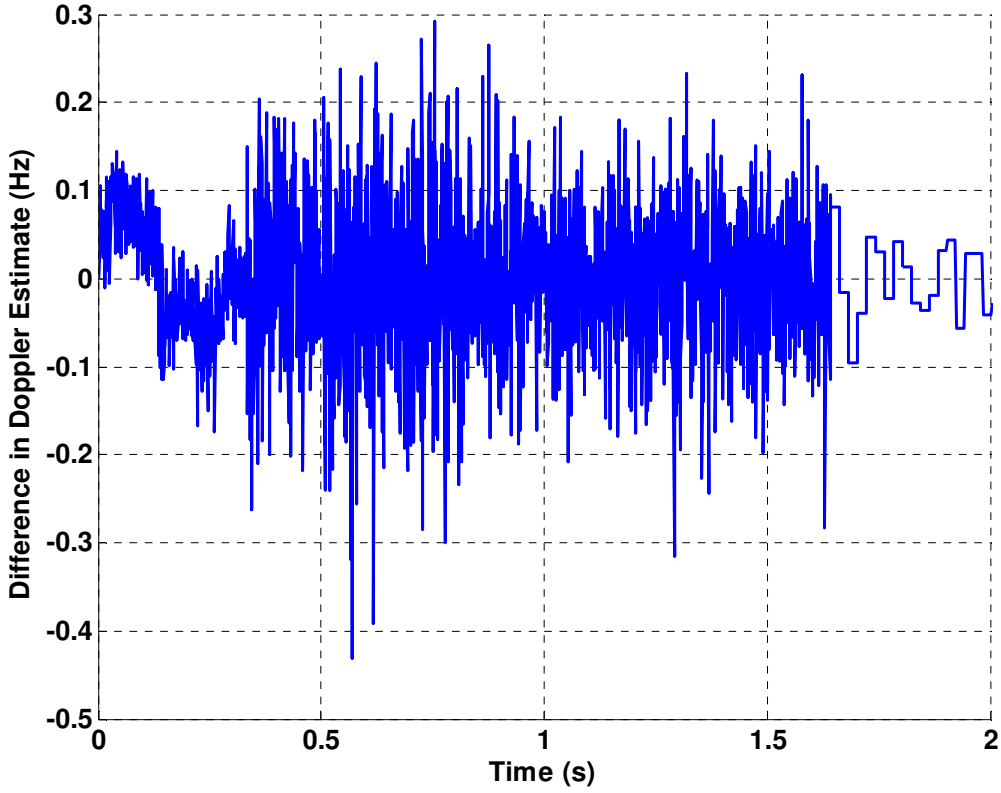

Figure 23 -- Difference in Doppler Estimates between Old and New Receiver when Tracking PRN 23

Figure 24 shows the Phase Locked Indicator (PLI) for the two receivers when tracking PRN 23. Like the Doppler, PLI results are identical to within the numerical approximation error.

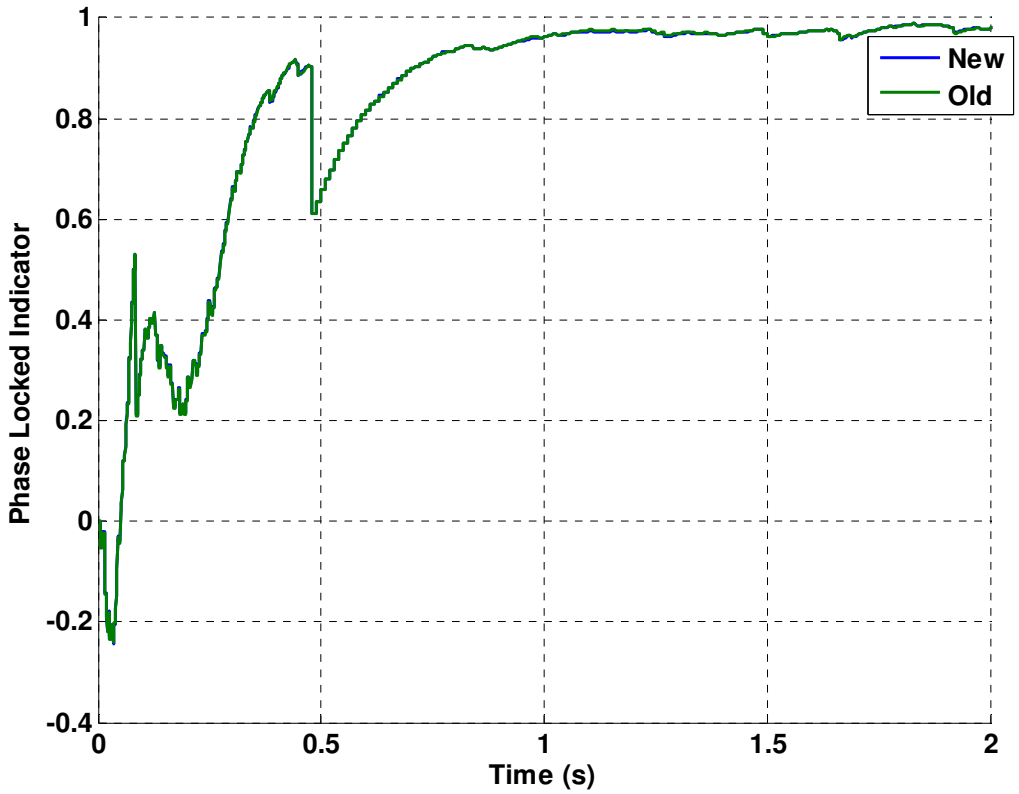

Figure 24 -- Phase Locked Indicator for Old and New Receiver when Tracking PRN 23

Tracking results verify that the multiplication and reduction components of the new DRC module function correctly.

Finally, Figure 25 shows the error of the GSNRx™ generated least-squares navigation solution of the dataset. This point-by-point solution uses seven code-phase measurements to calculate position and time.

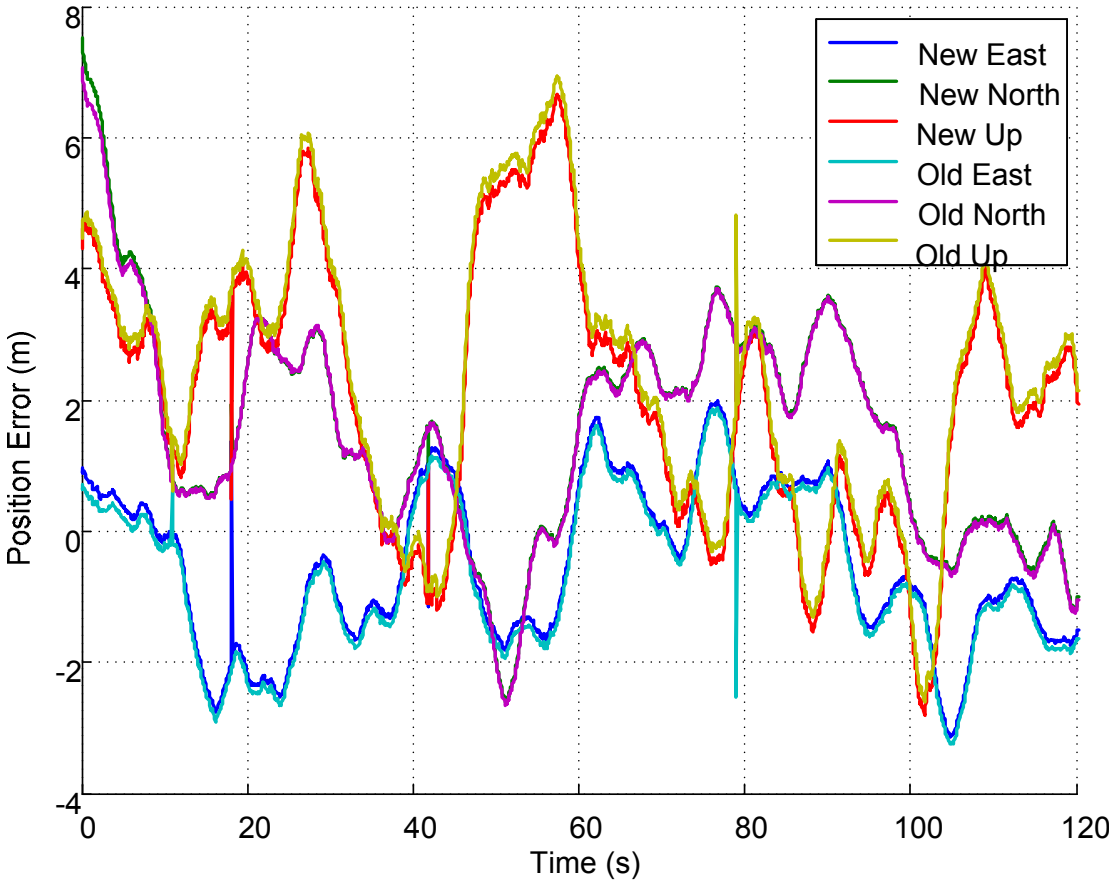

Figure 25 – Least Squares Navigation Solution Error

What is seen here is that there is a small, non-constant bias in the new navigation solution. This is a small bug in the software which has not yet been fixed due to time constraints.

There are two possible causes of the problem which need further investigation. First, the new correlation technique, where a correlation is performed over an entire code epoch – see Section 3.3.2.2– introduced a need for measurements to be propagated by the amount of time the channel clock lags / leads the receiver clock. The bias could be introduced through an error in propagating channel times epoch to epoch.

Next, the correlation operation itself may be causing the bias. In order to be able to isolate the DRC module from the post-processing of the receiver, the old DRC module was modified to allow asynchronous measurement generation. Even though the tracking metrics shown previously look very similar, if examined against the asynchronous module, a slight difference can be observed. Figure 26 shows the difference in the code phase estimate for identical I/Q samples when tracking PRN 23.

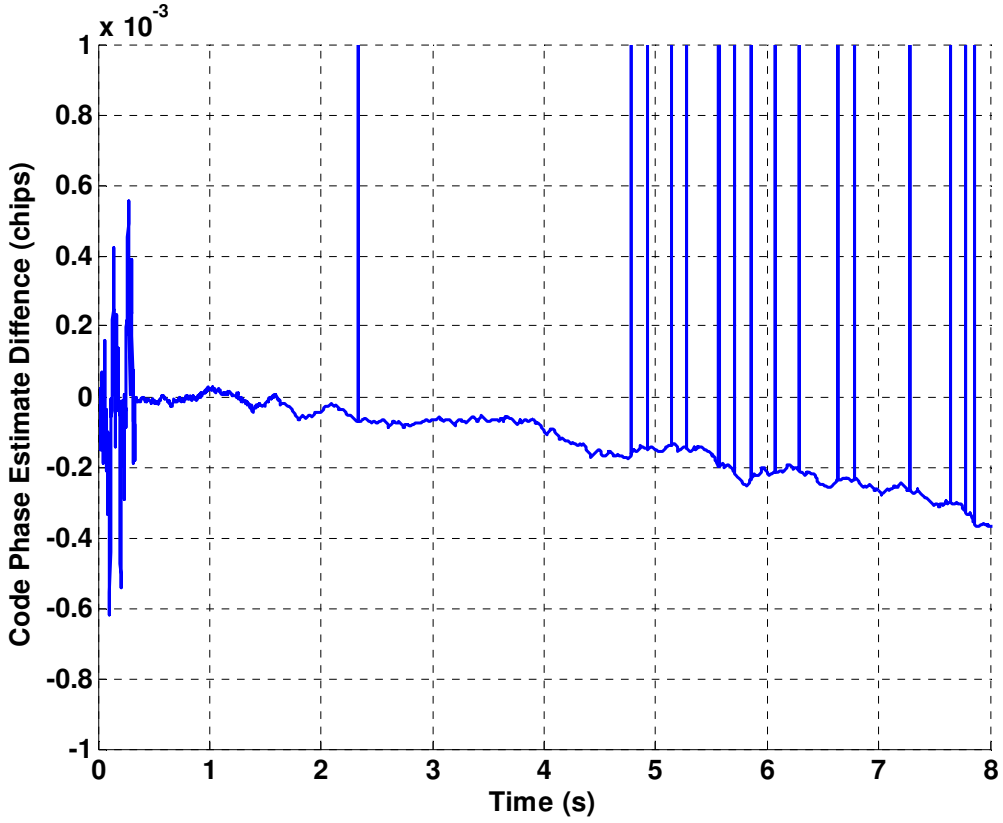

Figure 26 -- Code Phase Estimate Difference Between GPU and CPU Receiver for PRN 23

The jumps seen in the figure are not important, as they represent one module's code-rollover occurring before the others. What is notable, however, is the trend in the difference between the code phase estimates. As can be seen in Figure 27, the trend is not constant and appears to be a random walk.

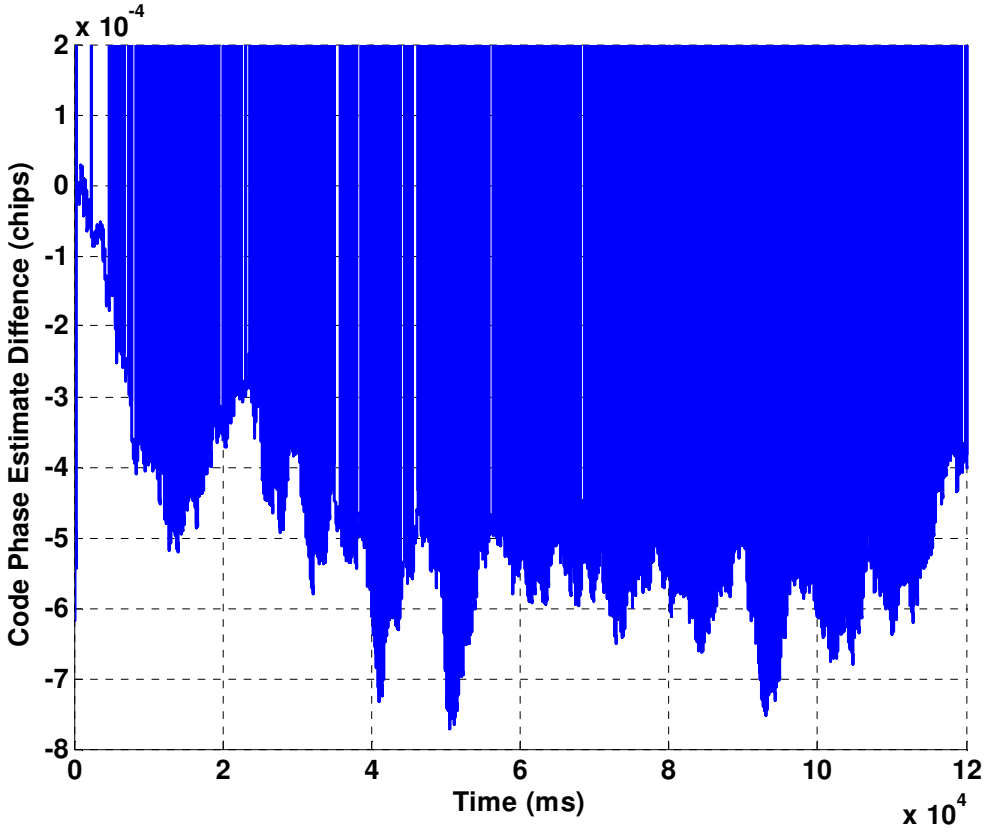

Figure 27 -- Code Phase Estimate Difference Between GPU and CPU Receiver for PRN 23

The code phase estimate differences shown in this figure are representative of all signals being tracked. A code phase estimate error of 400 μchips is equivalent to about 12 cm in pseudorange as per the following:

$$400\mu chips \bullet \frac{1s}{1.023x10^6 chips} \bullet \frac{299792458m}{s} \approx 12cm \qquad (5.1)$$

.

This is in line with the disparity in the least squares position solution. However, this value is not in line with the differences in the pseudoranges – see Figure 28 where the difference in the estimated pseudoranges in the two receivers is plotted against time. The pseudoranges are computed by each receiver in the measurement generation block as per:

$$PR = (Trx - Ttx) \bullet c \tag{5.2}$$

where:

PR is the pseudorange,

Trx is the received time and

Ttx is the transit time (calculated from the propagated code phase)

Since the channels operate asynchronously (see section 3.3.2.2), on a measurement epoch, the code phase needs to be propagated to a specific point in time, using the current channel time, the current code phase and the code phase rate. Thus, the larger error in the pseudoranges as opposed to the code phase can be attributed to an error in the channel time.

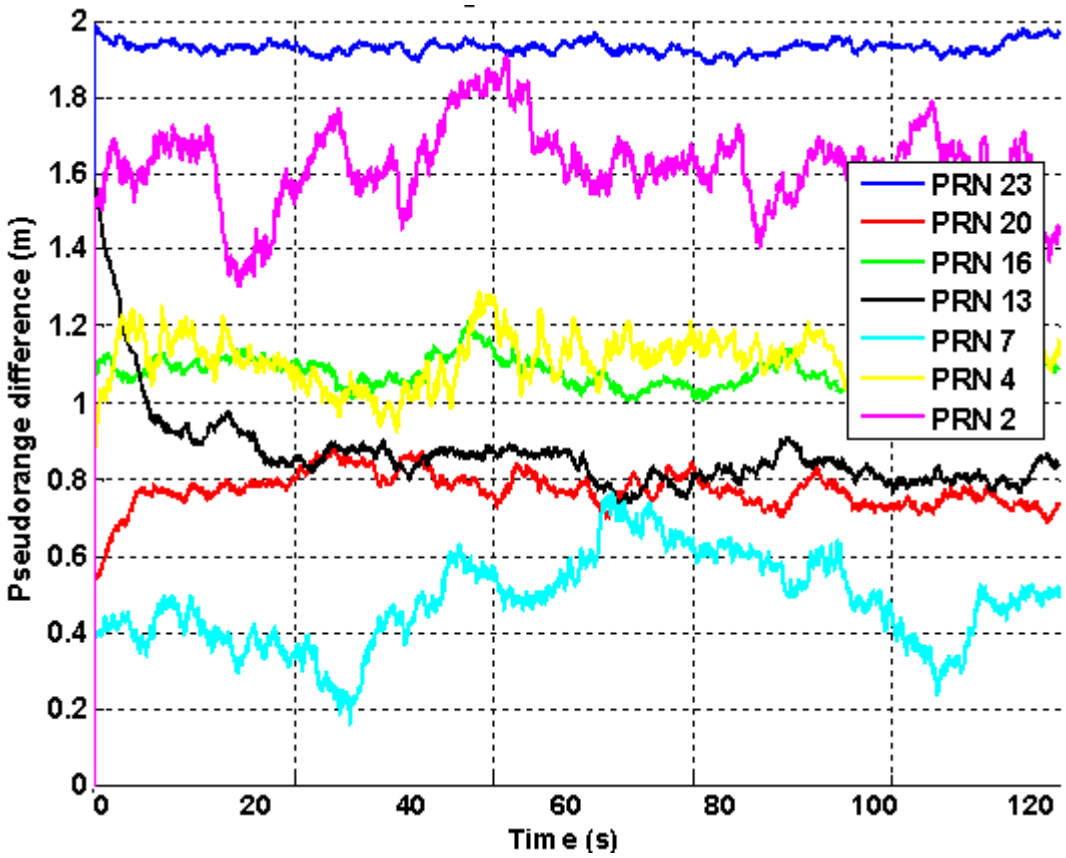

Figure 28 - Difference in Pseudorage Esimates between GPU and CPU receiver

A direct comparison of the disparity in pseudoranges versus the disparity in the code phase estimate can be seen in Figure 29. In the figure, the blue graph represents the negated difference in the estimated code phase (represented in metres), the red graph represents the difference in pseudorange and the magenta graph represents the difference in pseudoranges biased such that it equals the difference in pesudoranges.

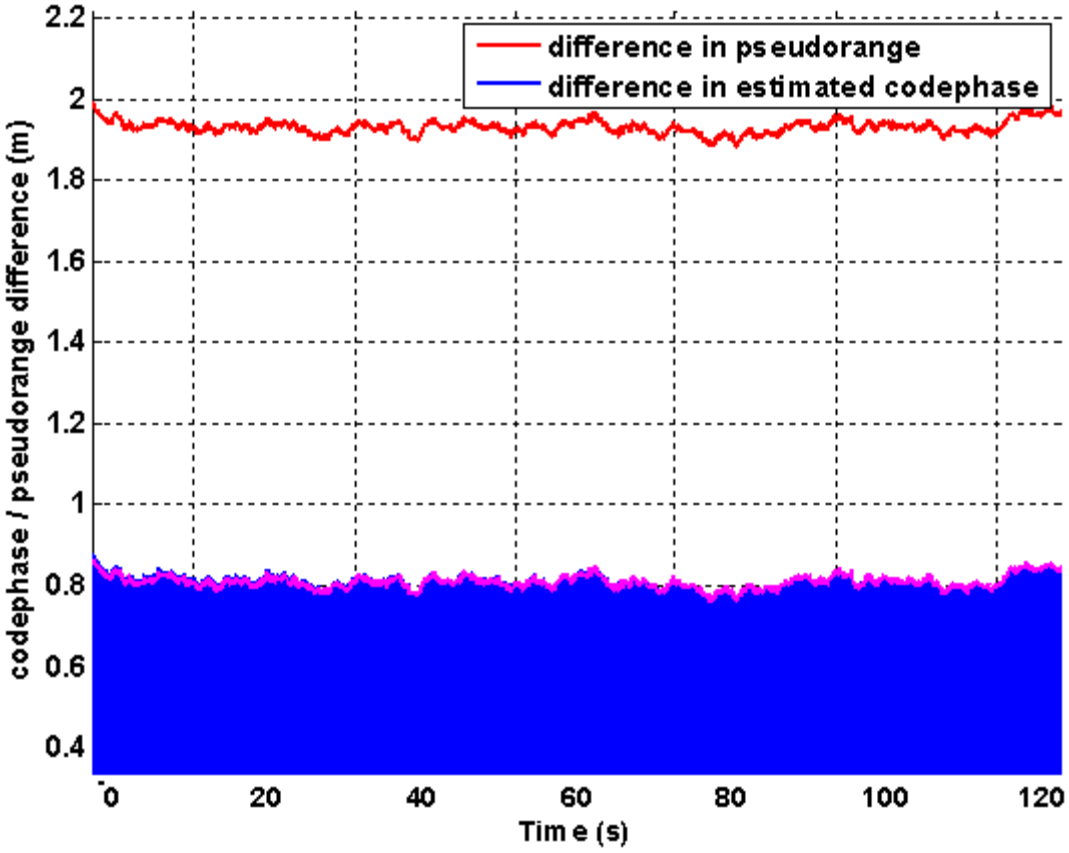

Figure 29 - Difference in Pseudorange Estimate and Code Phase Estimate between CPU and GPU – PRN 23

The magenta line is included to emphasize the fact that the difference in pseudoranges is a biased difference in code phase estimates.

Figure 30 repeats the above analysis with a different satellite – PRN 7.

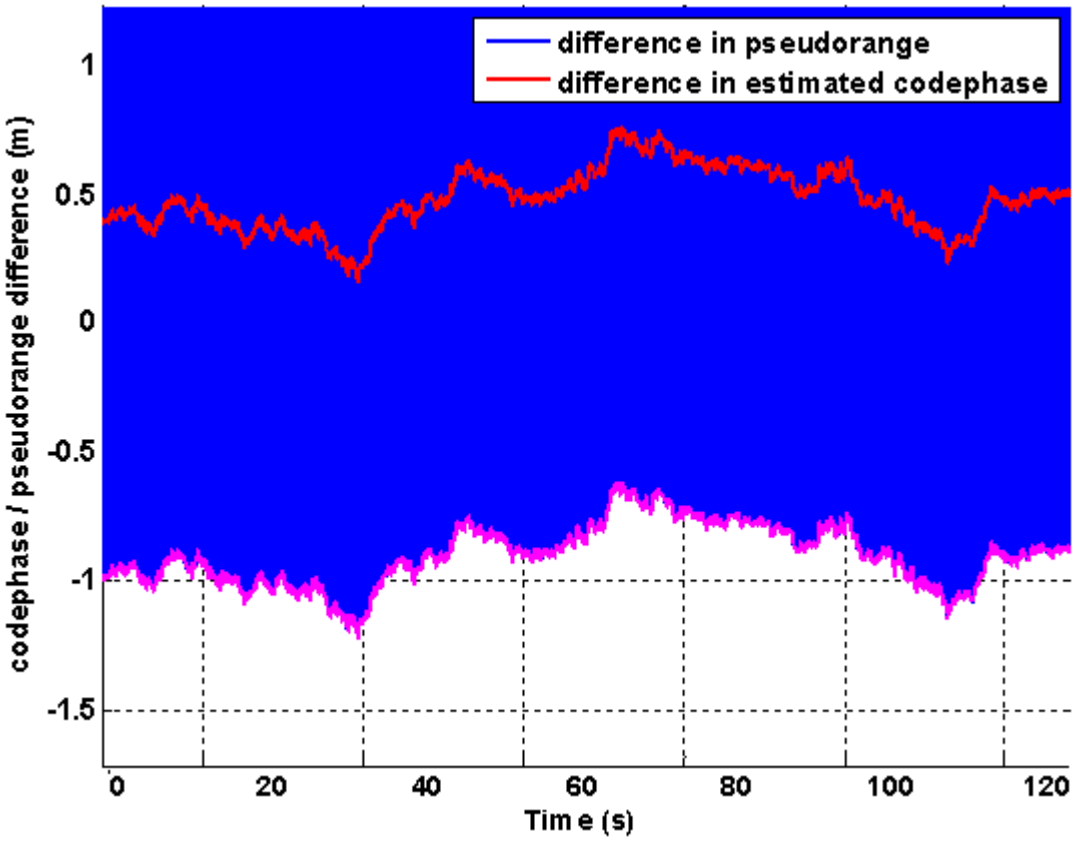

Figure 30 - Difference in Pserudorange Estimate and Code Phase Estimate between CPU and GPU -- PRN 7

The bias in the two cases is different – 1.12 m for PRN 23 and 1.38 m for PRN 7. When all available satellites are analyzed, this bias does not seem to be Doppler dependant.

The fact that the code phase estimate is not the only factor yielding different position results can further be seen by examining Figure 31.

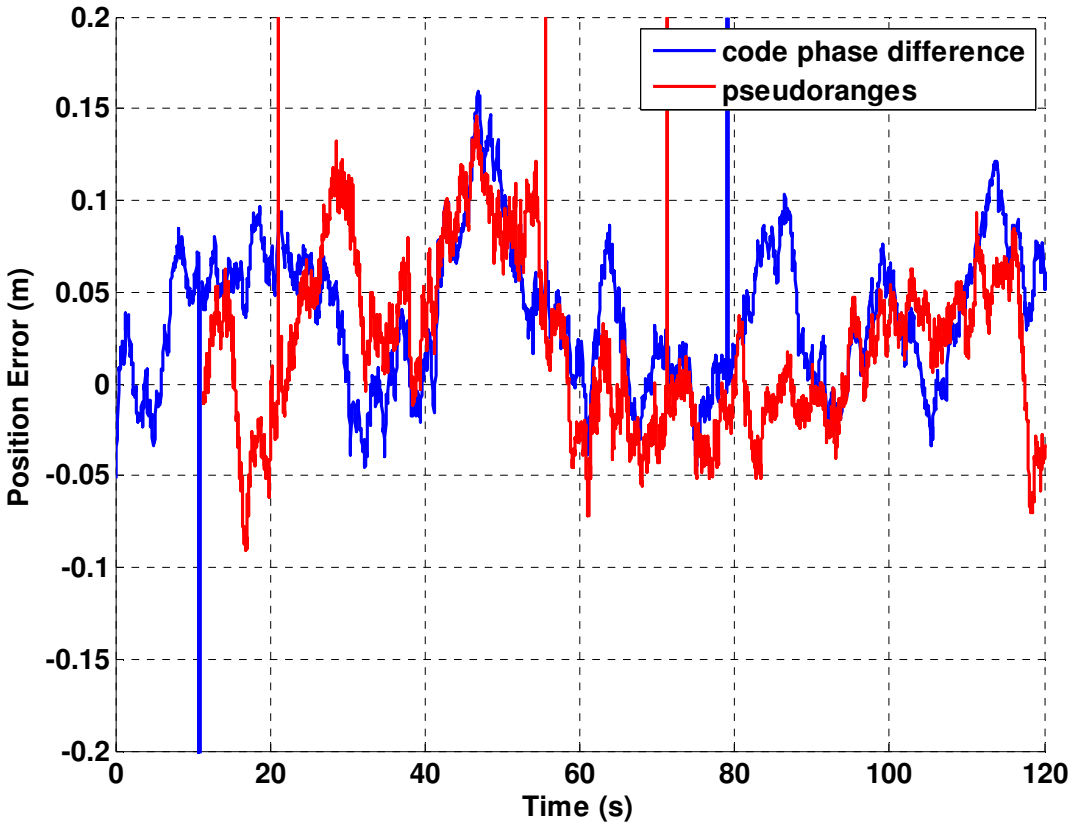

Figure 31 - Easting Position Disparity between Code Phase Estimate and Pseudorange Estimate

The above figure is the result of a least squares analysis on the difference in code phase estimates between the GPU and CPU receivers. The analysis was performed as follows:

- Ephemerides extracted by GSNRx^{TM} from the IF data were used to generate a point-by-point time series of satellite positions
- Satellite positions were used to generate a leas squares design matrix

- The ‘measurements’ provided as input to the least squares estimate were the difference in code phase estimates for each satellite
- A point-by-point least squares adjustment was performed
- Results were rotated to local-level frame, and plotted against difference in the least-squares position results generated by both receivers

The results do not match, as expected from the bias seen between the pseudoranges and the code phases.

As mentioned before, this bug has not yet been fixed and, for now, must be left as future work.

5.2.2 Timing Results

The profiler described in the previous section was used to obtain timing results and tune the number of blocks per grid and number of threads per block of the multiplication kernel. In the following test case, an 8-channel receiver operating at 40 Msps is considered.

Figure 32 shows the amount of time taken to correlate a 40 Msps signal over 1 ms for 8 channels with an early, prompt and late replica. The x-axis shows the number of threads per block used and the y-axis shows the time taken. Different lines represent the different

number of blocks per channel per grid used. The red, dotted line shows 1 ms, or real-time capability.

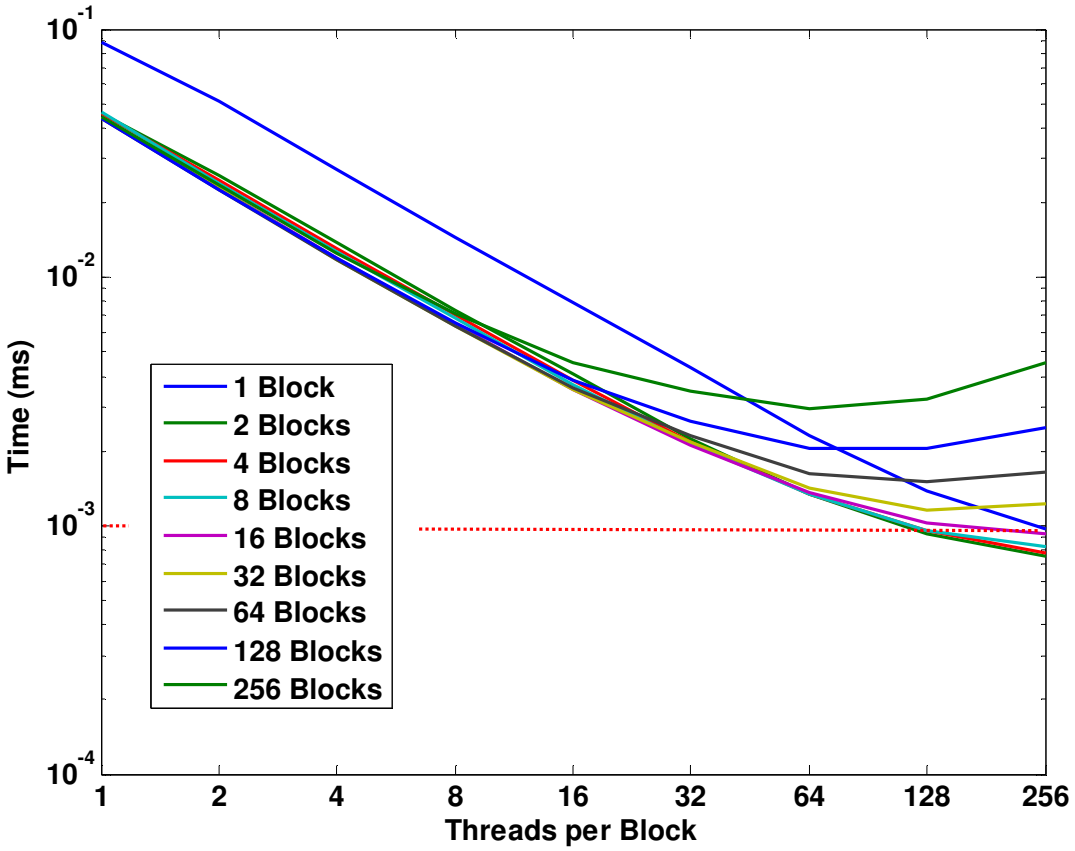

Figure 32 -- Time Taken to Correlate 1 ms of Data for Eight Channels

All but one case – one Block per channel – show similar results when thread-count is small. When only one block per channel is used, there are a total of only eight blocks, leading to only half utilization of the devices 16 multiprocessors. Initially, as thread count

is increased, execution time is decreased by a factor of the thread count, yielding a linear relationship between the two. This is optimal parallelization.

At some point, depending on block count, the device is fully utilized, thus threading overhead is increased, and there is less speedup per additional thread.

Of particular interest here is the portion of the graph below the red line, as this represents pseudo real-time operation; that is, it takes less time to process the data than to collect it – Figure 33 shows a zoomed inversion of this section of Figure 32.

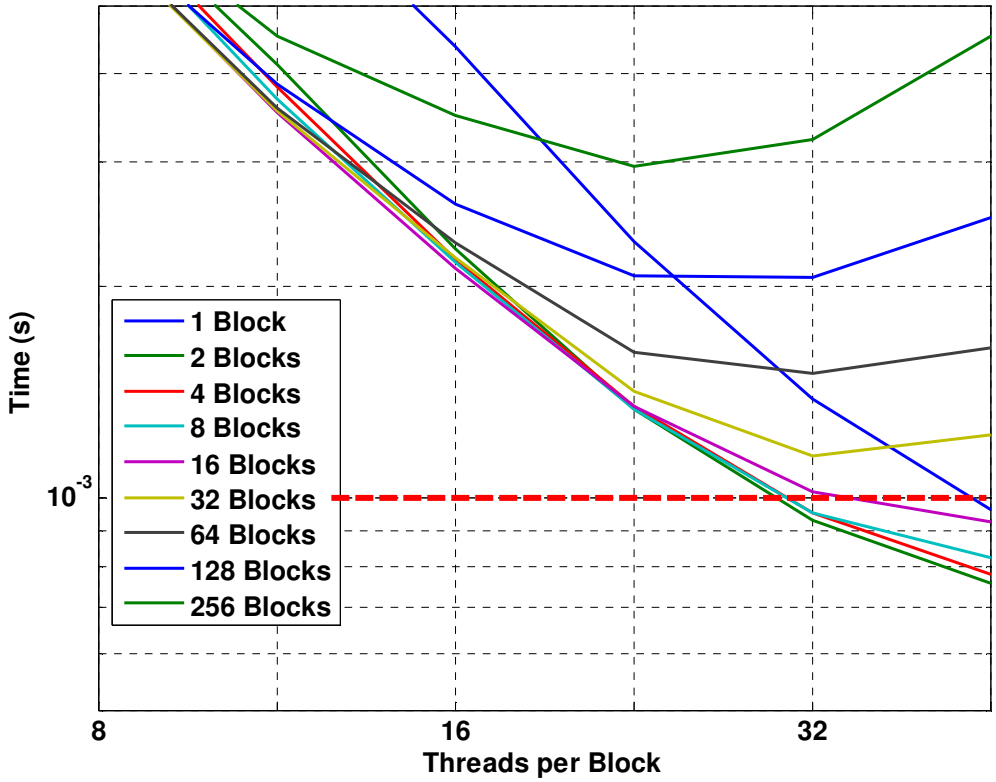

Figure 33 – Time Taken to Correlate 1 ms of Data For 8 Channels (zoomed)

Table 1 shows the break-down of the timing results for the optimal case: 2 blocks per grid per channel – with 8 channels, there are 16 blocks per grid total – and 256 threads per block. As there are 40,000 samples per epoch, this setup yields 80 samples to be processed by every thread, thus producing the sub-optimal stride discussed in section 3.3.2.3.

Table 3 -- Optimal Timing Results

Best Performance	
Blocks Per Grid Threads Per Block Samples Per Thread	2 256 80
Initiate Copy	1.6e-05 s
Receiver Processing Wait for Copy Completion Multiply Kernel Reduce Kernel	3.4e-04 s 2.0e-06 s 3.6e-04 s 2.4e-05 s
Copy Out Results	1.2e-05 s
Total Time	7.6e-04 s

“Initiate copy” is the time required to start a copy of IF samples onto the device. This is done as soon as correlation for the previous epoch is complete, before the receiver processing. Receiver processing includes all tasks the receiver does other than DRC, namely the medium and low rate tasks. “Wait for copy completion” is the amount of time the kernel has to wait before executing since data has not yet been copied onto the board. “Multiply” and “reduce kernel” show the times required for the multiply and reduce kernel respectively. Finally, “copy out results” shows the time required to copy the results back from the GPU.

5.2.3 Real-Time Results

The final test performed was real-time operation. Under the conditions described in the previous section – thread and block count – the module is capable of real-time operation over eight channels at 30 Msps.

While the receiver was capable of 40 Msps calculation, the front-end is only capable of producing and streaming either 30 or 50 Msps samples in real time. Thus real-time operation was limited to 30 Msps.

The operation of the receiver in real-time mode is identical to operation in post-process Ethernet mode. In the latter, data is collected on the NI, stored onto a RAID array and later transferred to the processing PC via Ethernet for processing. In the former, data is collected on the NI, stored temporarily in RAM then transferred via Ethernet for processing. In both cases, the processing PC accepts packets as they are streamed by the sample provider, stores them in the circular buffer and processes.

The receiver was allowed to run overnight to ensure robust operation.

Chapter Six: Conclusions

The following sections outline the conclusions and recommendations for future work from this thesis.

6.1 Conclusions

The following work has been accomplished and conclusions made throughout the progress of this thesis:

1) An FPGA DRC module has been developed for the Xilinx Vertex 4. Xilinx IP cores are used to instantiate input and output FIFOs and carrier and code NCOs. Dual clocked FIFOs are used in order to cross clock boundaries thus allowing all internal designs to operate on a single clock. The design, in its current state, is capable of single channel operation. In order to be completed, logic needs to be generated to accept control sequences from the host PC and an FPGA processing manager for GSNRx^{TM} needs to be created.

2) A GPU DRC module has been developed for an NVIDIAGeForce 8800GTX GPU. The module is divided into three parts: copying of data, multiplication and reduction. Copying deals with ensuring IF samples are available on the GPU hardware when required. It is optimized to limit the impact of high latency and

low bandwidth transfers. Multiplication is responsible for carrier wipe-off and correlation on a sample level. It is optimized to minimize the mathematical processing and memory load of the GPU. Finally, reduction handles adding the results of the multiplication section. It is optimized to minimize bank conflicts and maximize parallel usage of hardware. This design proved that a high-bandwidth – 40 Msps – real-time, software GNSS receiver can operate on consumer GPU and CPU hardware.

3) GSNRx™ has been modified to allow asynchronous measurements. Instead of stopping correlation on each channel on every measurement epoch, correlation is allowed to proceed until a code roll-over and measurements are propagated forward or back on a per channel basis. While this adds complexity as each channel must maintain its own clock, it allows uninterrupted processing of entire epochs. This doubles the speed in which the GPU can process data, and may also lead to speedups in future parallel designs.

4) A multi-threaded sample source has been developed. It is responsible for collecting samples from a front-end, storing them in a circular buffer and providing them to the receiver. It controls program flow between sample source – the hardware front-end – and sample sink – the receiver by allocating the tasks to

different threads and using blocking OS calls to retard threads which are not ready to run. It serves as the basis for all real-time GSNRx™ operation.

5) An interface to the SiGe front-end was developed to allow real-time operation utilizing the SiGe hardware. Aside from the multi-threaded sample source and modification of the driver – to increase stability – this development also included changing the firmware on the device itself to disable certain limiting aspects of operation, and to allow easier initialization. This module was part of the initial real-time design for GSNRx™. It proved that a real-time software based receiver can operate on a consumer PC running a standard operating system.

6) An Ethernet front-end was developed to allow sample providing and processing hardware to be housed on separate PCs. The software runs on two separate PCs: a sample provider and a sample user. A connection between the two is made on start-up and data is streamed via the LAN or Internet. Data is buffered on both sides, thus either can experience an interruption in processing and still be able to maintain real-time operation. This module helped show that over-the-network real-time processing is possible.

7) Functionality and timing results are presented for the GPU DRC module and the Ethernet front-end. The system is capable of 40 Msps operation and has been

tested real-time at 30 Msps. A bug remains in the software and is described in detail in Chapter Five.

6.2 Future Work

Based on the results and the experience gained throughout this work, the following recommendations are made for future work on this project:

1) Isolate and fix the bias seen in the navigation solution. As discussed in Chapter Five, this bias is most likely due to either the time-propagation required by each channel because of the new, asynchronous receiver, or improper code generation in the DRC module. Even though many tests have been performed to attempt to isolate the problem, it is difficult to separate processing noise from the error due to its small size.

2) Enhance the GPU DRC module to allow longer PRN codes, thus providing modernized signal processing capability. A new scheme of compressing, storing and retrieving the chips will need to be created for very large codes.

3) Modify the GPU DRC module to allow more than three correlators per channel. This will allow greater tracking flexibility and allow processing of modernized GNSS signals.

4) As hardware becomes available in the lab, optimize the GPU DRC module newer hardware. This will include, amongst other things, a transition from 32-bit floating point to 64-bit floating point arithmetic.

5) Enhance the capability of the FPGA DRC to allow multi-channel operation and control sequences from GSNRx^{TM}. The GPU processing manager can be modified as its structure is similar to what would be required by the FPGA. The asynchronous structure of the receiver should be maintained.

References

Akos, D.M., P.L. Normark and P. Engle (2001) "Real-Time GPS Software Radio Receiver," *Proceedings of ION NTM 2001,*Long Beach, CA, USA

Anghileri, M., T. Pany, D. Sanroma Guixems, J. Won and A. Ayaz (2007) "Performance Evaluation of a Multi-frequency GPS/Galileo/SBAS Software Receiver," *Proceedings of ION GNSS 2007*, 25-28 Septemner 2007, Fort Worth, TX.

Borre,K, D.M. Akos, N. Bertelsen, P. Rinder ans S.H. Jensen (2007) *A Software Defined GPS and Galileo Receiver*, Birkhauser Boston, New York, NY

Charkhandeh, S. (2007) *X86-Based Real Time L1 GPS Software Receiver*, M.Sc. Thesis, Geomatics Engineering, University of Calgary

Dovis, F., M. Spelat, P. Mulassano and C. Leone (2005) "On the Tracking Performance of a Galileo/GPS Receiver Based oh Hybrid FPGA/DSP Board," *Proceedings of ION GNSS 2005*, 13-16 September 2003, Long Beach, CA.

Danish GPS Centre (2010) *SoftGPS Project Homepage*, http://kom.aau.dk/project/softgps/ , last accessed February 16, 2010

Engel, F., P. Mumford, K. Parknison, C. Rizos and G. Heiser (2004) "An Open GNSS Receiver Platform Architecture," in *Journal of GPS 2004*, Vol. 3, No. 1-2, 63-69.

Free Software Foundation (2007) *The GNU General Public Licence*, http://www.gnu.org/licenses/gpl.html, last accessed February 23, 2010

Hobiger, T., T. Gotoh, J. Amagai, Y. Koyama and T. Kondo (2009) "A GPU Based Real-Time GPS Softwa Rereceiver" in *GPS Solutions* Volume 14, Number 2, August 2009

Horn, D (2005) "Stream Reduction Operations for GPGPU Applications" in *GPU Gems 2,* Addison Wesley, ch. 36, pp 573-589

Harris, M (2007) *Optimizing CUDA,* NVIDIA Corp, 88 pages

Hennessy, J.L. and D.A. Patterson (2007) *Computer Architecture a Quantitative Approach, Fourth Edition*, Morgan Kaufmann Publishers, San Francisco, CA, USA

Intel (2009) *Intel Processors by Family,* Retrieved 5 July 2009, from http://ark.intel.com/?iid=processors_body+resources_ark.

Jamieson, P. (2007) *Improving the Area Efficiency of Heterogeneous FPGAs with Shadow Clusters*, PhD thesis, Department of Electrical Engineering, University of Toronto, Canada.

Jouppi, P.N. (2007) "Memory Hierarchy Design" [Chapter 5] in *Computer Architecture a Quantitative Approach, Fourth Edition*, Morgan Kaufmann Publishers, San Francisco, CA, USA

Kaplan, D.E and C.J. Hegarty (2005) *Understanding GPS: Principles and Applications*, Artech House Publishers, Norwood MA

Knezevic, A., C. O'Driscoll and G. Lachapelle (2010) *Co-Processor Aiding for Real-Time Software GNSS Receivers*. Proceedings of International Technical Meeting, Institute of Navigation (San Diego, 25-27January), 12 pages.

Lachapelle, G., (2008) *Advanced GNSS Theory and Applications,* ENGO 625 Course Notes, Department of Geomatics Engineering, University of Calgary, Canada

Lackey, R.J. and D.W. Upmal (1995) *Speakeasy: The Military Software Radio*, IEEE Communications Magazine, May 1995, p 56-62

Ledvina, B. M. Psiaki, S. Powell and P. Kittner (2003) "A12 Channel Real-Time GPA L1 Software Receiver," *Proceedings of ION GNSS 2003*, Aneheim, CA.

Luebke, D. (2007) *The Democratization of Parallel Computing*, NVIDIA Research, 13 pages

McCool, M. D. (2001) *SMASH: A Next Generation API for Programmable Accelerators*, Computer Graphics Lab, Department of Computer Science, University of Waterloo, 31 pages.

MinGW.org (2009) *MinGW Minimalist GNU for Windows*, http://www.mingw.org/, last accessed February 23, 2010

msdn (2010) *QueryPerformanceCounter Function*, http://msdn.microsoft.com/en-us/library/ms644904(VS.85).aspx, last accessed February 23, 2010

NVIDIA (2009) *CUDA Programming Guide*, Version 2.2, NVIDIA Corp, 137 pages

O'Driscoll, C. and M. Petovello (2007) *GSNRx – Algorithm Design Document*, PLAN group, Calgary, 77 pages.

Owens, D., D. Luebke, N. Goivinaraju, M. Harris, J. Kruger, A. Lefohn and T. Purcell (2007) "A Survey of General-Purpose Computing on Graphics Hardware," in *Computer Graphics Forum*, Volume 26, number 1, pp 80-113.

Pany, T. F. Forster and B. Eissfeller (2004) *Real Time Processing and Multipath Mitigation of High-bandwidth L1/L2 GPS Signals with a PC-Based Software Receiver*, Institute of Geodesy and Navigation, University FAF Munich, Germany, 15 pages.

Pany, T., S. Moon, M. Irsigler and B. Eissfeller (2003) "Performance Assessment of an Under Sampling SWC Receiver for Simulated High-Bandwidth GPS/Galileo Signals and Real Signals," *Proceedings of ION GNSS 2003*, 9-12 September 2003, Portland, OR.

Parkinson, K., A. Dempster, P. Mumford and C. Rizos (2006) "FPGA Based GPS Receiver Design Considerations," in *Journal of GPS 2006*, Vol. 5, No. 1-2, 70-75.

Petovello, M.G. and G. Lachapelle (2008) "Centimeter-Level Positioning Using an Efficient New Baseband Mixing and De-Spreading Method for Software GNSS Receivers," *Journal on Advances in Signal Processing (JASP)*, In Press.

Petovello, M., C. O'Driscoll, G. Lachapelle, D. Borio, H. Murtaza (2008) "Architecture Benefits of an Advanced GNSS Software Receiver," in *Journal of GPS 2008*, Vol. 7, No. 2, 156-168.

Roger, D., U. Assarsson and N. Holzschuch (2007) "Efficient Stream Reduction on the GPU" in *Workshop on General Purpose Processing on Graphics Processing Units*, oct 2007

Schamus, J., J. Tsui and D. Lin (2002) "Real-Time Software GPS Receiver" *Proceedings of ION GNSS 2002*, September 2003, Portland, OR.

SDCC (2009) *SDCC – Small Device C Compiler*, http://sdcc.sourceforge.net/, last accessed February 23, 2010

ste_meyer (2010) *libusb-win32*, http://sourceforge.net/projects/libusb-win32/files/libusb-win32-releases/, last accessed February 23, 2010

Thor, J., P. Normark and C. Stahlberg (2002) "A High-Performance Real-Time GNSS Software Receiver" *Proceedings of ION GNSS 2002*, September 2003, Portland, OR.

University of Colorado at Boulder (2010) *GNSS @ Colorado Center for Astrodynamics Research*, http://ccar.colorado.edu/gnss/, last accessed February 22, 2010

Volodya (2002) *fx2_programmer*, http://volodya-project.sourceforge.net/fx2_programmer.php, last accessed February 23, 2010

Viterbi, A (1995) *CDMA: Principles of Spread Spectrum Communications,* Addison-Wesley Publishing Company

Ward, P (2005) "Satellite Signal Acquisition, Tracking and Data Demodulations," [Chapter 5] in *Understanding GPS Principles and Applications* Artech House Publishers, Norwood MA

Won, J., T. Pany and G. Hein (2006) *GNSS Software Defined Radio, Real Receiver or Just a Tool for Experts* , Inside GNSS, July/August 2006 p 48-56

Zhang, W (2008) *Portable and Scalable FPGA-Based Acceleration of a Direct Linear System*, MSc Thesis, Department of Electrical Engineering, University of Toronto, Canada.