12-2019

GPU-Accelerated Demodulation for a Satellite Ground Station

Emily Clark Young
Utah State University

GPU-ACCELERATED DEMODULATION FOR A SATELLITE GROUND STATION

by

Emily Clark Young

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical Engineering

Approved:

Jacob Gunther, Ph.D. Major Professor	Todd Moon, Ph.D. Committee Member
Reyhan Baktur, Ph.D. Committee Member	Richard S. Inouye, Ph.D. Vice Provost for Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2019

ABSTRACT

GPU-Accelerated Demodulation for a Satellite Ground Station

by

Emily Clark Young, Master of Science

Utah State University, 2019

Major Professor: Jacob Gunther, Ph.D.
Department: Electrical and Computer Engineering

One consequence of the increasing number of small satellite missions is an increasing demand for high data rate downlinks. As the satellites transmit at high data rates, ground-side receivers need to demodulate the transmitted data as quickly as possible. While application specific hardware can be designed, software defined radio solutions for ground stations are attractive for their flexibility, adaptability, and portability.

Another industry trend is the increasing use of Graphics Processing Units (GPUs) in general-purpose processing. By performing many operations simultaneously, GPUs are capable of accelerating processing when given a problem that can be implemented in a parallel manner. Furthermore, once a parallel algorithm is implemented, further speedups are possible by increasing hardware resources without need for any revision in the algorithm.

This project combines the above ideas by implementing a software defined radio algorithm to quickly demodulate high-speed data on a GPU. It demonstrates the viability of the GPU in software defined radio applications and particularly in the area of fast demodulation.

(84 pages)

PUBLIC ABSTRACT

GPU-Accelerated Demodulation for a Satellite Ground Station

Emily Clark Young

One consequence of the increasing number of small satellite missions is an increasing demand for high data rate downlinks. As the satellites transmit at high data rates, ground-side receivers need to demodulate the transmitted data as quickly as possible. While application specific hardware can be designed, software defined radio solutions for ground stations are attractive for their flexibility, adaptability, and portability.

Another industry trend is the increasing use of Graphics Processing Units (GPUs) in general-purpose processing. By performing many operations simultaneously, GPUs are capable of accelerating processing when given a problem that can be implemented in a parallel manner. Furthermore, once a parallel algorithm is implemented, further speedups are possible by increasing hardware resources without need for any revision in the algorithm.

This project combines the above ideas by implementing a software defined radio algorithm to quickly demodulate high-speed data on a GPU. It demonstrates the viability of the GPU in software defined radio applications and particularly in the area of fast demodulation.

To my late great-grandmother, Joye S. Peterson, a real-life “Rosie the Riveter;” you have always been my hero.

ACKNOWLEDGMENTS

I would like to express my thanks to a few of the teachers and educators who have been influential in my life: to Dr. Gunther, for his guidance and encouragement which made this project possible; to Mr. Rippon, my 8th grade Geometry teacher, for enabling me to see what I was capable of; and to Dr. Moon, who first introduced me to communications, information theory, and signal processing, and has been an excellent teacher.

I am also deeply grateful for the continual love and support of my parents, in-laws, and most of all my husband Caleb, who is also my best friend.

Emily Young

CONTENTS

Page

ABSTRACT ii

PUBLIC ABSTRACT iii

ACKNOWLEDGMENTS v

LIST OF TABLES viii

LIST OF FIGURES ix

ACRONYMS x

1 INTRODUCTION 1
1.1 Motivation 1
1.2 Thesis overview 3

2 REVIEW OF LITERATURE 4
2.1 Synchronization in communications systems 4
2.1.1 Timing and phase synchronization 4
2.1.2 Carrier frequency offset 9
2.2 Software defined radio 12
2.3 Implementations of software defined radio 13

3 RESEARCH AND DESIGN METHODS 15
3.1 Signal model 15
3.2 Timing and phase correction by complex kurtosis 16
3.2.1 Timing offset correction 17
3.2.2 Phase offset correction 18
3.3 Phase offset correction by “min/max” equalization method 19
3.4 Carrier frequency offset correction by spectral estimation 21
3.5 System verification 22

4 CUDA IMPLEMENTATION AND METHODS 23
4.1 GPU Architecture 23
4.2 Indexing 23
4.3 Parallel Reduction 25
4.4 Methods of convolution 28
4.4.1 Inner-product Convolution 29
4.4.2 Multi-threaded convolution 29
4.4.3 Fast Convolution 31
4.4.4 Conclusions 31

5 RESULTS 33
5.1 Optimal synchronization parameters 33
5.2 Comparison of phase synchronization methods 35
5.3 CUDA Optimizations 37
5.3.1 Symbol block size 37
5.4 Timing tests 38

6 CONCLUSION 41
6.1 Future work 42

REFERENCES 44

APPENDICES 50
A CODE LISTINGS 51
A.1 C Functions 51
A.2 GPU Functions 58
B System Diagrams 70

LIST OF TABLES

Table Page

4.1 Run times for inner-product convolution method 29

4.2 Run times for multi-threaded convolution method 31

4.3 Run times for fast convolution method 31

5.1 GPU execution time for phase correction with 100 tests 36

5.2 Average runtime in milliseconds by packet length. 37

5.3 Data rates in Mbps based on block and grid size. 38

5.4 Code Timing Tests 39

5.5 Demodulation runtime for 500-symbol packets of data. 40

LIST OF FIGURES

Figure Page

2.1 QPSK constellation with sample timing offset. 5

2.2 QPSK constellation with phase offset. 6

2.3 Received QPSK constellation with carrier frequency offset. 9

2.4 Spectrum of the transmitted signal in the passband. 10

2.5 Spectrum of demixed signal with a frequency offset. 10

2.6 Spectrum of signal at baseband after CFO correction. 11

3.1 Simulation of modulated signal 16

3.2 Kurtosis as a function of timing offset. 18

3.3 Box fitted around received symbols. 19

3.4 Box fitted around rotated symbols. 20

3.5 Spectrum of $(y[n])^4$ 22

4.1 CUDA Memory hierarchy 24

4.2 Parallel reduction for a single block 26

5.1 Receiver BER by number of matched filters 34

5.2 Receiver BER by number of phase points tested. 35

5.3 Receiver BER by method of phase correction 36

B.1 Full System Diagram 71

B.2 Timing Correction System Diagram 72

B.3 Carrier Frequency Offset Correction Diagram 73

B.4 Phase Correction System Diagram 74

ACRONYMS

ASIC	application-specific integrated circuit
BER	bit error rate
CFO	carrier frequency offset
CPU	central processing unit
DC	direct current
DICE	Dynamic Ionospheric CubeSat Experiment
FFT	fast Fourier transform
FPGA	field programmable gate array
GMSK	Gaussian minimum shift keying
GNSS	global navigation satellite system
GPS	global positioning system
GPU	graphics processing unit
LDPC	low-density parity-check
OFDM	orthogonal frequency division multiplexing
QPSK	quadrature phase shift keying
PLL	phase locked loop
SDR	software defined radio
SNR	signal to noise ratio
TED	timing error detector
UAV	unmanned aerial vehicle

CHAPTER 1

INTRODUCTION

1.1 Motivation

In October 2011, the Dynamic Ionospheric CubeSat Experiment (DICE) project was launched into space. DICE consisted of two 1.5U CubeSats, each equipped with the instrumentation to measure plasma density, electric field, and magnetic field. Previous to the launch of DICE, CubeSat missions relied primarily on transmission over amateur radio bands, and downlink speeds were relatively low. Between the limited overpass time of the satellite and the large amount of data that needed to be transmitted to the earth, the DICE mission required a higher than average downlink rate – on the order of megabits per second. A satellite radio was developed that could transmit at the high data rates, and the satellite was licensed to operate on a governmental frequency. From the satellite side, the problem of fast data transmission was solved [1].

However, on the earth side, the downlink rate was too high to be demodulated in real time on a PC, and the data was instead recorded and demodulated at a later time [2]. This gave rise to the need for ground station demodulation that is fast, portable, and flexible. Previously available solutions only needed to handle data rates on the order of kilobits per second, but as small satellites grow in usage there will be an increased need to demodulate larger quantities of data in a limited amount of time. A review of recently launched CubeSats [3] indicates that this is the case: early satellites transmitted in the kilobits per second range, while more recently launched satellites may transmit megabits per second.

While the increase in CubeSat missions necessitates the ability to handle more and more data, it also necessitates flexible and complex communications systems. Software defined radio (SDR) is a compelling option with significant benefits. SDRs are versatile enough to

handle any modulation scheme without requiring custom hardware for each scheme. They are potentially less expensive than a hardware radio, since most or all of the modulation and demodulation can be done on any processor. They can be updated remotely to accommodate additional standards, which could be especially beneficial for spacecraft radio systems. Furthermore, changes and updates don't require any hardware rework and no lead time to implementation.

Although there exist many compelling reasons for using software defined radios, they are still subject to specific challenges. One such challenge, particularly relevant in the CubeSat case presented above, is data rate. Custom hardware can be made to operate at very high speeds, while software applications have some intrinsic overhead that slows processing. To counteract the slower processing speeds, software has to be optimized for the hardware it is developed on, but complex designs can make this task difficult. One possible approach is the use of Graphical Processing Units (GPUs) for software radio applications. GPUs can offer substantial speedups compared to CPUs, provided that the algorithm can be implemented in a parallel manner due to the inherent parallel architecture of the GPU: while a CPU may be able to execute a few software threads at once, a GPU can execute thousands. Furthermore, once a parallel algorithm has been designed, further speedups may be achieved by simply increasing the hardware resources, without necessarily having to change the algorithm.

The GPU approach has some significant limitations. GPUs typically consume a substantial amount of power. Commercially available embedded GPUs could easily use 10 W of power; in contrast, an FPGA used for the same application may operate on less than 0.5 W. As a result, it would be difficult to produce a small, deliverable product using a GPU when the same functionality can be implemented on an FPGA or an ASIC with much lower power usage.

That said, in the right setting GPUs are still a powerful and useful computational tool. Power consumption is much less of a concern for a ground station application than for a satellite or mobile communication system. When programmed using a C-extended language

such as OpenCL or CUDA, floating point computations are generally trivial to implement. This effectively eliminates the problem of fixed point quantization noise. In contrast, an FPGA must explicitly allocate hardware resources to handle floating point operations.

As GPUs are seeing further improvement and development, it is worth studying their effectiveness in a variety of engineering applications. This project aims to demonstrate the viability of using GPUs in software radio applications, specifically in performing demodulation.

1.2 Thesis overview

The rest of this paper is organized as follows: Chapter 2 presents a review of the literature surrounding time, phase, and frequency synchronization, which is the main challenge in demodulation. It further considers the literature regarding software defined radio, especially in satellite applications, and discusses the hardware platforms used to implement software defined radios. Chapter 3 outlines the signal model and algorithmic approach used in this project. Chapter 4 discusses the implementation of the algorithm on the GPU, as well as some methods uniquely suited to the GPU architecture. Results are presented in Chapter 5. Chapter 6 provides conclusions and suggestions for further work.

CHAPTER 2

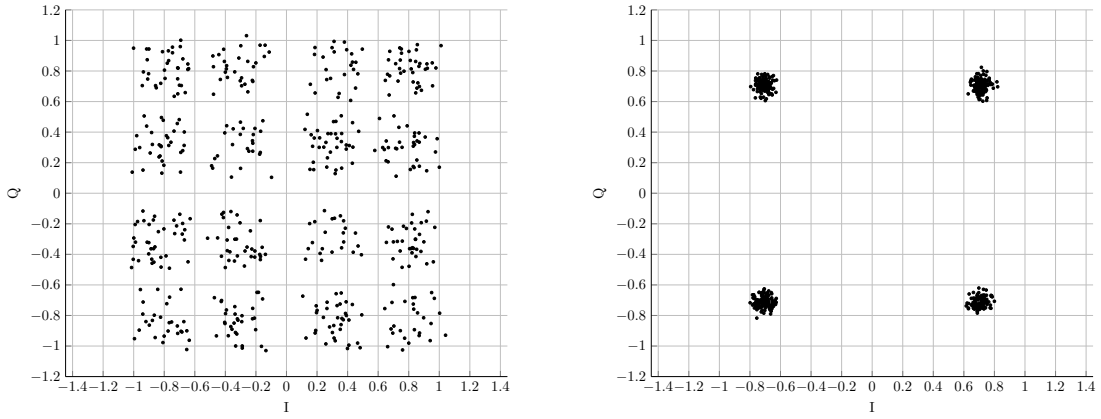
REVIEW OF LITERATURE

2.1 Synchronization in communications systems

Communications systems are subject to many non-ideal channel effects. These include sample timing offset, phase offset, and carrier frequency offset. Here, a brief study is made of the many synchronization methods available for use in a digital communications system.

2.1.1 Timing and phase synchronization

Timing and phase synchronization are common issues in receiver systems. Sample timing offset occurs when the incoming signal is not sampled at the optimal symbol time. This has the effect of "scattering" the symbol constellation points, moving them closer to the symbol decision boundaries and increasing the risk of decision error. This scattering effect is illustrated in Figure 2.1. Figure 2.1a shows the effect of a large, uncorrected timing offset, with the received symbols appearing to spread out from their correct values. Conversely, Figure 2.1b shows the received symbols with a corrected sample timing offset, and the received symbols are closely clustered around their intended values. Both figures represent received constellations in the presence of very little noise, so that the spreading effect is due to the sample timing offset. At lower SNR levels, the timing offset spread will result in higher bit error rates since small noise levels will be sufficient to "push" symbols into incorrect decision regions.



(a) Received QPSK constellation with sample timing offset.

(b) Received QPSK constellation with corrected sample timing offset.

Fig. 2.1: QPSK constellation with sample timing offset.

Phase offsets are rotations on the incoming data which can incorrectly project symbols into the wrong decision regions. This is demonstrated in Figure 2.2: the symbols are received with a 45 degree phase offset, and are rotated to move the constellation points to the correct place. Pre-rotation, the symbol clusters may lie on or near decision region boundaries, as shown here. Then, even very small amounts of noise can cause symbols to be projected into the wrong decision regions.

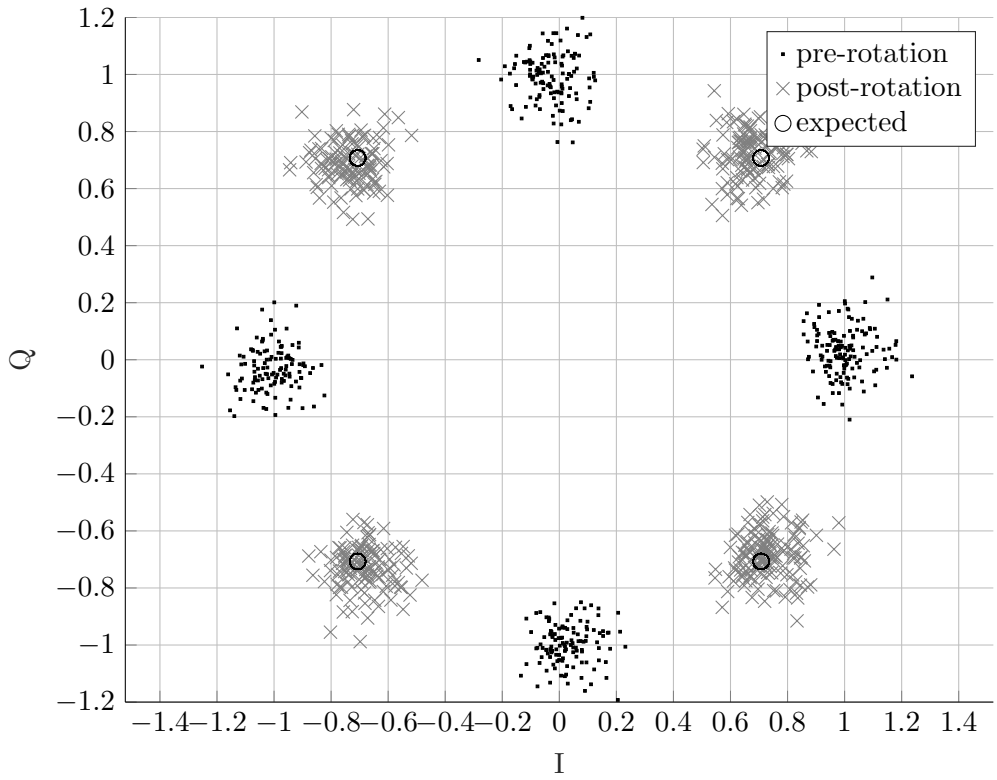


Fig. 2.2: QPSK constellation with phase offset.

One of the earliest and still frequently used approaches to timing and phase correction is the use of Phase Locked Loops (PLLs). PLLs use an error detector, loop filter, and an oscillator in a feedback configuration to detect and track phase and frequency in an incoming signal. The error detector signals that a timing or phase correction is needed. An interpolation scheme is used to adjust the sample time or rotate the symbols based on the error signal. The system operates continually to compensate for the detected error signal [4, Appendix C].

The PLL recovery approach has been well studied. For phase recovery, PLLs may use one of several different timing error detectors (TEDs) have been introduced, adapted, and extended over the years. These include the Gardner TED [5], the zero-crossing TED [6], the maximum likelihood TED [7], and the Mueller and Müller TED [8].

A simple interpolation scheme introduced by Farrow [9] is frequently used in demodulation systems. More recently, various timing recovery interpolation schemes have been introduced, such as the trigonometric polynomial interpolation in [10]. An optimal interpolation filter is described in [11]. Interpolation is also sometimes performed using a multirate filtering approach as in [12].

PLL-based implementations are often used in demodulating continuous data streams. It takes some nonzero amount of time for the system to lock onto the signal and for the phase error to diminish [4, p 730]. Due to the acquisition time, PLLs are not a good approach in packetized burst communications. In a burst-mode system the PLL would have to lock onto the signal for each packet, and some symbols at the beginning of each packet would inevitably be lost.

Burst mode communications can be split into two categories: "data aided" and "non-data aided." In the data aided case, packets are sent with a known preamble which is used to recover the symbol timing. Data aided algorithms generally outperform non-data aided algorithms at the expense of wasted bandwidth.

Data aided timing and phase recovery algorithms are plentiful in the literature. The authors of [13] demonstrate the usage of a known preamble to recover timing by selecting the timing that maximizes the correlation of the pilot data with the input signal. It essentially performs a filtering operation, with the filter being formed from the pilot data. The algorithm presented in [14] is able to recover the timing and phase simultaneously with a shorter preamble than other techniques. In [15], the authors propose a maximum-likelihood algorithm which estimates the carrier frequency offset, corrects for it, and then calculates the timing and phase offset using closed form expressions, which are based on an optimized preamble. The optimized preamble allows for a less complex receiver system. In [16], an FFT is performed on the preamble data of a GMSK signal. The timing is estimated from the FFT phase at a certain frequency and the carrier frequency estimate is obtained from the DC portion of the FFT. The estimated parameters are used during the preamble period to compensate for the offsets, following which the phase offset is coarsely estimated.

Although the performance of data-aided synchronization algorithms are generally superior, there are important motivations for using blind synchronization. Sending a preamble in each packet adds overhead and reduces overall data throughput. Blind synchronization is thus motivated in part by interest in high speed data transmission.

One of the earliest and most commonly used blind estimators is introduced by Oerder and Meyr in [17]. The timing offset estimation is performed by computing the spectral component from the squared input signal. It requires that the signal be upsampled by a factor of 4. The estimator in [18] also uses an upsampling factor of 4 but maximizes the log-likelihood function to find the timing offset. The authors claim that it is similar in complexity to the Oerder-Meyr estimator [17], but performs with higher accuracy. A similar method is mentioned in [19] that uses an absolute value operation on the data as part of the estimation. The estimation algorithm in [20] uses only two samples per symbol and has similar performance to the conventional Oerder-Meyr. It does not claim to be the optimal solution in terms of accuracy, but it performs reasonably well with fewer samples per symbol than the other estimators [17–19] and does not require any expensive nonlinear computations.

A blind estimation algorithm for space-time block code systems is described in [21]. It finds the beginning of a block by measuring the correlation of multiple samples and finding the maximum correlation. The synchronization parameters are estimated using second order statistics. The algorithm requires the system to have two receiving antennas. The paper suggests that the algorithm is an effective synchronization scheme for space-time block codes which may be useful in areas such as software defined and cognitive radios.

In [22], the timing and carrier frequency offsets are extracted from the cyclic correlation of the input data, made possible because of the cyclostationarity of OFDM recieved signals. Both parameters are derived from the phase of the cyclic correlation, which is not affected by the impulse response of the transmitting channel. Therefore it is not necessary to know anything about the channel to recover the timing and carrier frequency using the proposed estimator.

The non-data aided class of algorithms described by Moeneclaey and Bastele [23] builds off of the Mueller and Müller [8] and Gardner [5] error detectors in a feedback configuration, and only requires one sample per symbol. The approach is modified for a feedforward configuration in [24].

2.1.2 Carrier frequency offset

Carrier frequency offset (CFO) is a nonideality that occurs at the demixing step in the receiver. If the receiver demixes with a frequency that is not perfectly matched to the signal carrier, the signal is not completely brought to baseband. This occurs either because the transmitter and receiver oscillators are not exactly the same, or due to a Doppler frequency shift [25]. In either case, the result is a "spinning" effect on the symbols and the system performance is degraded. This effect is demonstrated in Figure 2.3. A QPSK signal was transmitted, but received with a carrier frequency offset such that the symbols don't seem to cluster together at all.

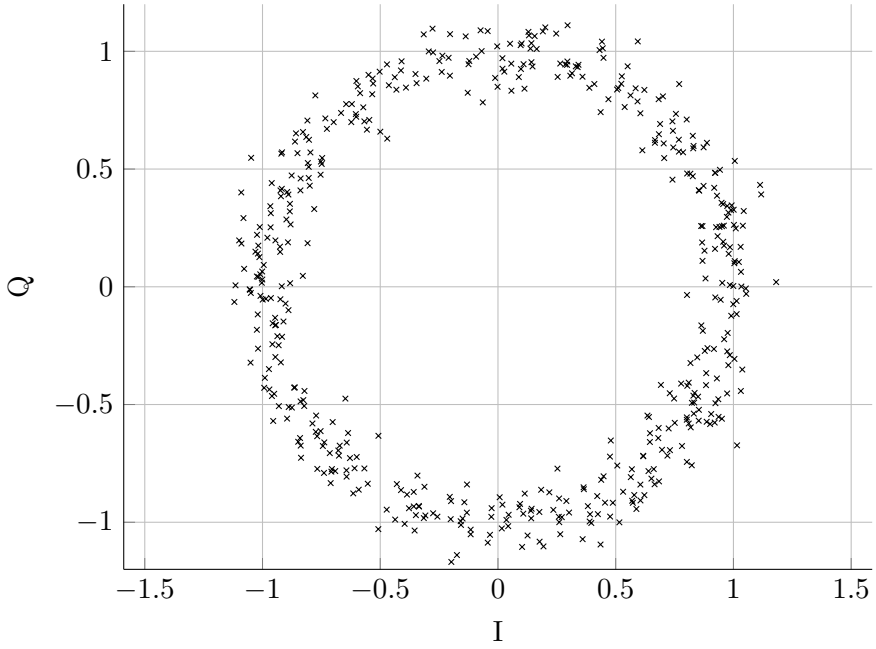


Fig. 2.3: Received QPSK constellation with carrier frequency offset.

Furthermore, OFDM signals rely on the division of a frequency band into orthogonal "subchannels" [26]. The presence of an uncorrected frequency offset can therefore cause co-channel interference.

The de-mixing frequency, f_r, is equal to the carrier frequency f_c plus an offset v. In this scenario the de-mixed signal, $r[n]$, is equal to the transmitted signal multiplied by a complex exponential

$$r[n] = s[n]e^{j2\pi\nu n}$$

Figure 2.4 shows the magnitude spectrum of the transmitted signal at its carrier frequency. Figure 2.5 shows the magnitude spectrum of the signal after de-mixing. Figure 2.6 shows the magnitude spectrum of the signal at baseband.

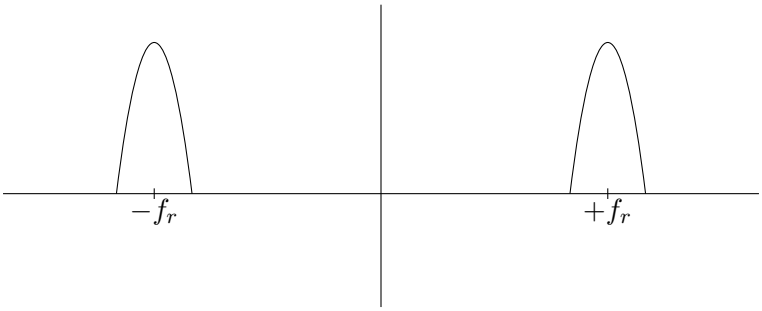


Fig. 2.4: Spectrum of the transmitted signal in the passband.

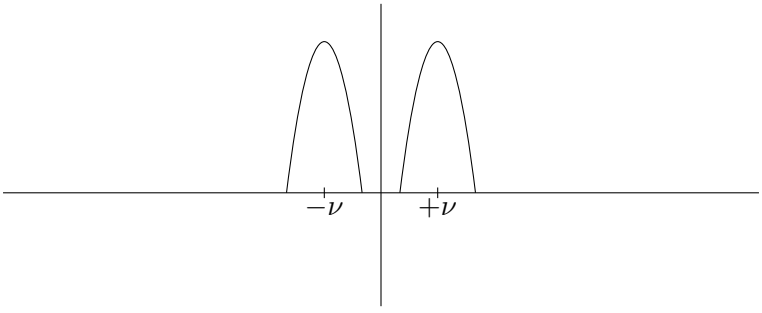


Fig. 2.5: Spectrum of demixed signal with a frequency offset.

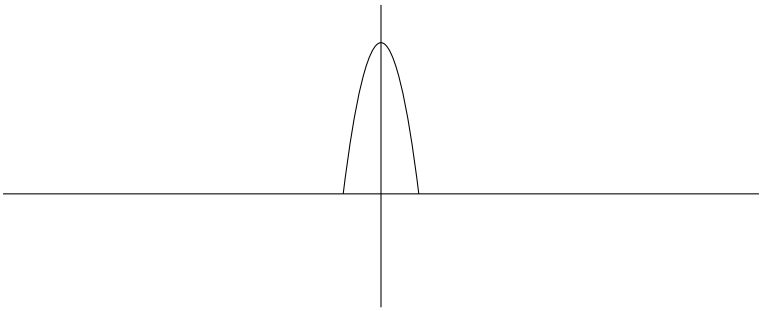


Fig. 2.6: Spectrum of signal at baseband after CFO correction.

In the continuous data stream case, as long as the offset is not too large, and the noise bandwidth parameter is chosen appropriately, the frequency offset can often be corrected by the PLL at the same time as the phase correction.

Several of the burst mode synchronization algorithms mentioned in the previous section perform timing and carrier frequency correction simultaneously. For example, the cyclic correlation method in [22] estimates the timing and carrier frequency offsets jointly. Carrier frequency correction was also described in the data-aided methods in [15] and [16].

Other papers describe algorithms specifically used for frequency offset correction. Several are reviewed in [27], and mentioned here. The Rife-Boornstyn estimator [28] is a maximum likelihood method which performs a coarse and then a fine search for the frequency offset. The well known Luise-Regiannini estimator [29] is a maximum likelihood based strategy that maximizes the autocorrelation with respect to the frequency mismatch. The Tretter [30] and Kay [31] estimators are least-squares methods that perform similarly to the optimal maximum likelihood estimators at high SNR, but perform poorly at low SNR. However, they are both linear algorithms with lower complexity than the Rife-Boornstyn estimator mentioned previously. Morelli and Mengali [32] present three estimation schemes based on known training data. More recently, the authors of [33] proposed two non-data aided methods to estimate frequency based on irregular symbol repetition. They claim to perform well at low SNR levels with a short burst duration.

In summary, the problem of timing, phase, and carrier frequency offset estimation and correction is well explored for both continuous and burst communications. The methods

differ in terms of performance and complexity and an approach must be carefully selected to match the constraints of a given problem.

2.2 Software defined radio

Software defined radio has seen considerable growth in the past twenty years or so, as improvements in processor capabilities have made it an increasingly viable option. Since their inception in the early 1990s, much work has been done to explore a wide range of software radio capabilities in many applications. For example, a software-defined radio based emergency vehicle alert system is presented in [34], software radio platforms for smart homes are evaluated in [35], and a software-based GPS receiver is described in [36]. Software defined radios are used in [37] as cognitive relays, which would be implemented on UAVs for ground stations to combat interference in satellite downlinks.

Software defined radio is a particularly attractive option in satellite systems, given their heavy reliance on reliable communication systems. Beginning in the mid-1900s, scientists envisioned a network of man-made communication satellites that would enable worldwide broadcast services [38]. Over the past several decades the space industry has largely been driven by communications-related interests, leading to the advent of global navigation (such as GPS), reliable long-distance and mobile communications, improved weather forecasting and more [39].

The need for advanced communications systems is not strictly limited to communications-based applications. Scientific satellite missions may employ cameras or sensors to collect data, which then needs to be relayed to an earth-side receiver. In fact, the telecommunications subsystem of a satellite is arguably one of the most important subsystems, since without a way to transmit or receive information, a satellite is essentially useless. The problem has continued to grow more complex as more small satellite missions are launched. Increased regulatory requirements, limited bandwidth, security measures, and sophisticated communications protocols drive the need for flexible, cost-effective communications solutions [40]. This is especially true for satellite "swarm" missions, where several satellites are sent into orbit to operate cooperatively and may communicate with each other as well as

with ground stations. Software radio systems can be programmed to handle a variety of modulation schemes, operate on a variety of transmission bands, and implement complex protocols. Software radios can also be updated remotely, allowing the satellite communications system to use new protocols or modulation schemes as needed.

The appeal of software defined radios on satellites is evident from its treatment in the literature. Examples include [40–42]. Software radios are also being explored for satellite ground station solutions. One such system was developed and tested in [43] as an alternative to dedicated hardware components for a GNSS-R system. The authors of [44] outline a software radio solution for ground stations that uses commercial off-the-shelf components. A similar system is implemented in [45] which cancels interference using a maximum likelihood algorithm, estimates Doppler shift, and addresses co-channel interference.

2.3 Implementations of software defined radio

Intrinsic in the design of a software radio system is the choice of hardware platform. Relevant considerations in that choice include power usage, speed, development cost, and flexibility. The systems in [34] and [36] are implemented using GNU Radio, which suggests implementation on a traditional processor. The authors of [46] present an approach to designing satellite payloads on an FPGA with a goal of flexibility and reconfigureability. A specific architecture for a satellite software defined radio is given in [47] for implementation on an FPGA. The architecture in [41] pairs an FPGA System on Chip (SoC) with a dual-core ARM processor. The contribution of [48] is a model-based software radio design for an FPGA, in which a complete communications system is implemented.

Of particular interest in this study is the usage of graphics processing units in software-defined radio. While FPGA and CPU based processing is widely used and sometimes preferred, the capabilities of GPU platforms have seen considerable progress in recent years and is therefore worth studying. Some publications that use GPUs for all or part of the signal processing are noted here.

The computationally expensive correlation process of the GNSS protocol is implemented in [49] using a GPU platform. Notable speedups were achieved with this implemen-

tation. A complete software-defined transceiver was implemented on a GPU in [50] which achieves near-maximum-likelihood detection and corrects interference using equalization techniques. The receiver implemented in [51] estimates relevant parameters by simultaneously correlating all possible combinations of sidelink synchronization signal sequences. A GPU-based software defined radio was also implemented in [52] and a study performed to evaluate the load placed on the CPU. A high-throughput OFDM system was implemented on a GPU in [13] and substantial speedups were observed.

Error correction can also be performed on GPUs; [53, 54] and [55] each perform LDPC with decoding with GPU-specific optimizations. Similarly, [56] uses a "divide-and-conquer" approach to perform Hamming distance calculations on each trellis in parallel.

CHAPTER 3

RESEARCH AND DESIGN METHODS

The bulk of the algorithm to be implemented in this project is based on the model introduced by [57]. It presents a method of timing and phase synchronization for a burst transmission system by using statistical measures on a block of signal data. The method is a low complexity approach that is easily adapted to run in parallel.

The proposed system is a realization of a software defined radio on a GPU platform using burst-mode synchronization techniques.

3.1 Signal model

It is assumed that the signal contains no training or pilot data, and the system processes the symbols in blocks or packets rather than as an infinitely long incoming sequence. The bit data for the simulation is generated randomly, differentially encoded, and modulated with a QPSK modulation scheme. Timing offset is simulated on the transmitter side by using a pulse-shaping filter with a random timing offset. Once the baseband signal $s(t)$ has been generated, it is mixed to a carrier frequency f_c. Frequency offset is simulated by adding a randomly chosen offset, ν, to the transmitted frequency. The receiver demixes the signal at the frequency $f_r = f_c + \nu$. A phase offset term ϕ between $-\pi/4$ and $\pi/4$ is chosen at random and added to the argument of the demixing sine and cosine waves as shown in (3.1) and (3.2);

$$I_r(t) = r(t)\cos(2\pi f_r t + \phi); \tag{3.1}$$

$$Q_r(t) = r(t)\sin(2\pi f_r t + \phi). \tag{3.2}$$

The baseband signal simulation is summarized by the diagram in Figure 3.1.

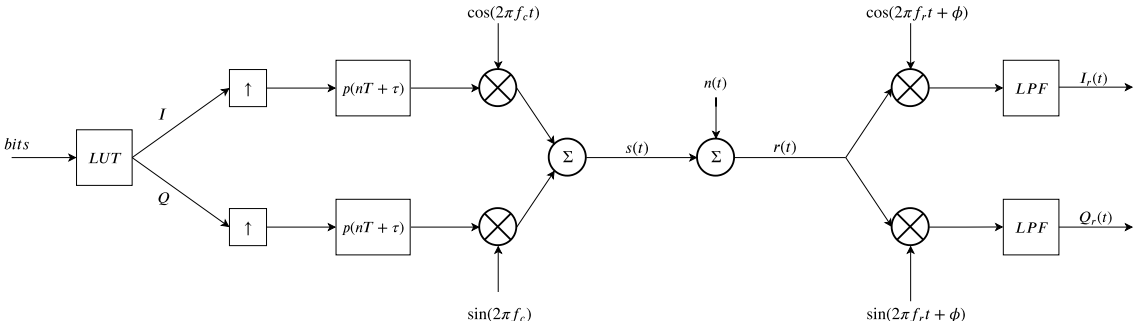


Fig. 3.1: Simulation of modulated signal

The effect of the channel is modeled by adding white Gaussian noise $n(t)$ to the transmitting signal.

$$r(t) = s(t) + n(t)$$

The noise is generated by the Box-Muller Transform [58]. The Box-Muller Transformation takes two independent samples, U_1 and U_2 of a uniform distribution, $\mathcal{U} \sim (0, 1)$. The uniform samples are provided by the rand() function in C. Then, two normally distributed samples, Z_1 and Z_2 are found by the following equations:

$$Z_1 = \sqrt{-2 * \ln U_1} \cos 2\pi U_2$$

$$Z_2 = \sqrt{-2 * \ln U_1} \sin 2\pi U_2$$

The samples Z_1 and Z_2 are scaled by σ_n. All of the noise samples are generated by this process, resulting in $n(t) \sim \mathcal{N}(0, \sigma_n^2)$.

Once the signal is modulated down to baseband and the in-phase and quadrature-phase portions of the signal have been extracted, the system performs timing, phase, and frequency corrections to the signal.

3.2 Timing and phase correction by complex kurtosis

Because of its parallel nature, a feedback or feed-forward structure such as a PLL is not an effective way to solve the timing offset or phase offset problems in a GPU implemen-

tation of a demodulator. Therefore, a statistical method is employed which is much more effective to implement in parallel. This system assumes that no training data is available and processes the data in blocks, rather than as an infinitely long incoming sequence.

The complex kurtosis is a second-order statistic that can be interpreted as a measure of the "Gaussian-ness" of a sequence [59]. The complex kurtosis is given by

$$K_c(y) = E\{y^4\} - 2(E\{|y|^2\})^2 - |E\{y^2\}|^2. \tag{3.3}$$

If y is a real variable, the kurtosis reduces to

$$K_r(y) = E\{y^4\} - 3(E\{y^2\})^2. \tag{3.4}$$

A kurtosis of 0 indicates a perfectly Gaussian distribution. By extension, a kurtosis far away from zero indicates a relatively non-Gaussian distribution.

3.2.1 Timing offset correction

Doing the matched filtering of the signal at a timing offset results in inter-symbol interference; effectively, it mixes together multiple symbols. By the central limit theorem [60, p. 390], when independent identically distributed variables are mixed the result tends toward a Gaussian distribution. Using the kurtosis as a measure of Gaussian-ness, it follows that the timing delay which results in the least Gaussian sequence is the delay that minimizes the timing error [57]. The timing problem is solved in this system by creating a bank of matched filters with various timing delays. The complex kurtosis is calculated on the sequence filtered by each of the matched filters. The minimum of these kurtoses values corresponds with the matched filter with the timing offset closest to the signal timing offset. The plot in Figure 3.2 shows the complex kurtosis of the sequence as a function of the timing offset.

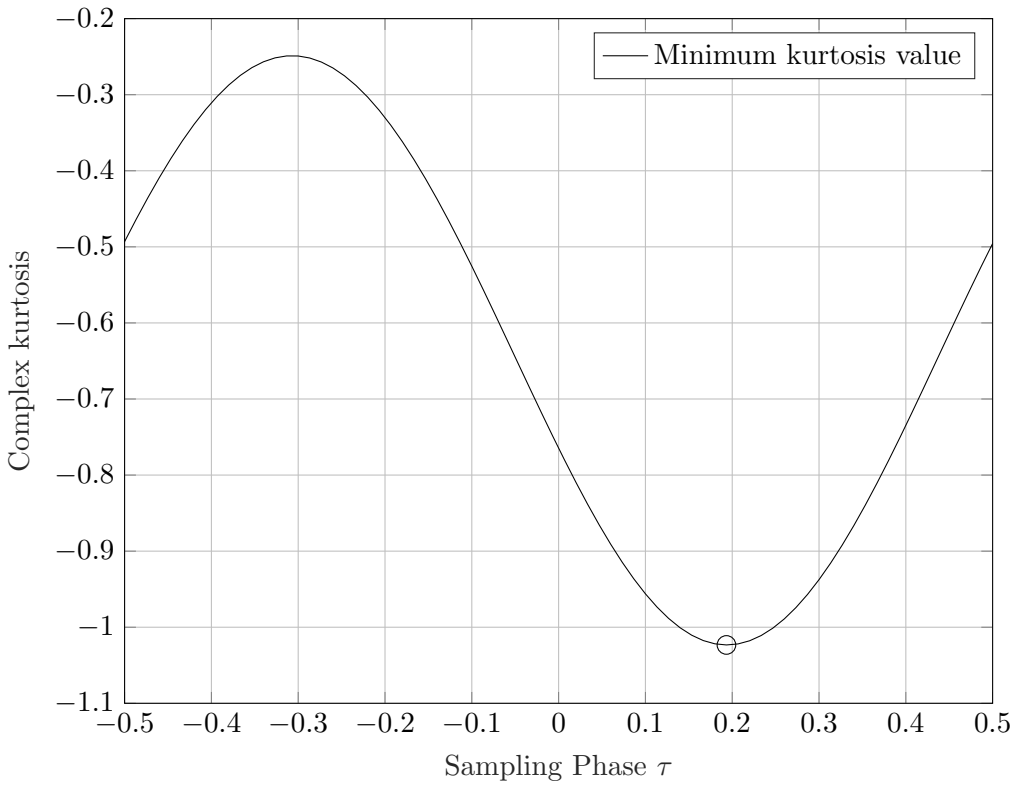


Fig. 3.2: Kurtosis as a function of timing offset.

3.2.2 Phase offset correction

Kurtosis-based phase offset correction uses a similar procedure to the timing offset correction. The downsampled, filtered symbols are rotated by an angle $\{\theta|\theta_1, \theta_2, ...\theta_{np}\}$ by multiplying the symbols by the rotation matrix R.

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

The phase resolution np defines the number of test phases used. The test phases are equally spaced between $-\pi/4$ and $\pi/4$. For each rotation, the real kurtosis is calculated on both the in-phase and quadrature-phase branches, then summed together. The rotation angle which results in a minimum kurtosis is taken to be the best correction. This method results

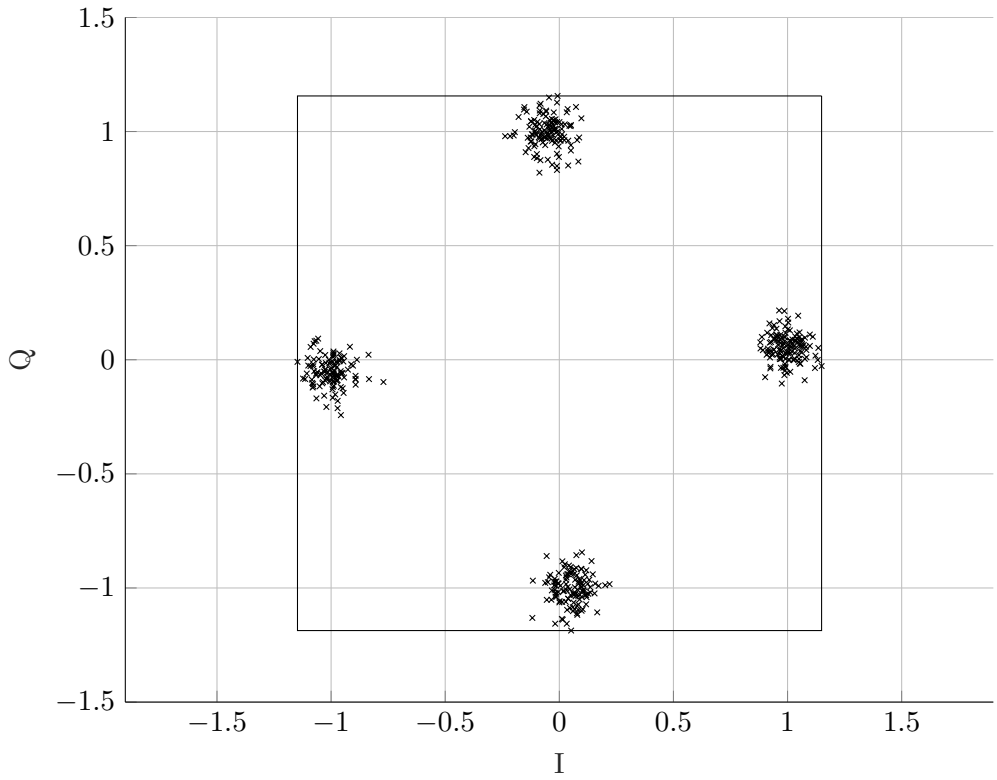


Fig. 3.3: Box fitted around received symbols.

in a $\pi/2$ phase ambiguity which is easily resolved by use of differential encoding.

3.3 Phase offset correction by "min/max" equalization method

A second phase correction method, the "min/max" method, was also implemented. The method is based on the idea of fitting a square box around the outside of the symbol constellation such that the box contains all of the symbols in the constellation [61]. The symbols are then rotated in such a way as to minimize the length of the sides of the square box. When a phase offset exists the symbols can take on more extreme values in the x- and y- directions, thus increasing the size of the box. Figures 3.3 and 3.4 illustrate this idea.

Note that the fitted box in Figure 3.3 is larger than the fitted box in Figure 3.4 due to the phase offset. When the symbols are correctly located, the box size is minimized. As in the kurtosis method, the symbols are rotated by several different test phases. For each test phase, the symbols are rotated and then projected onto the y-axis. The maximum value of

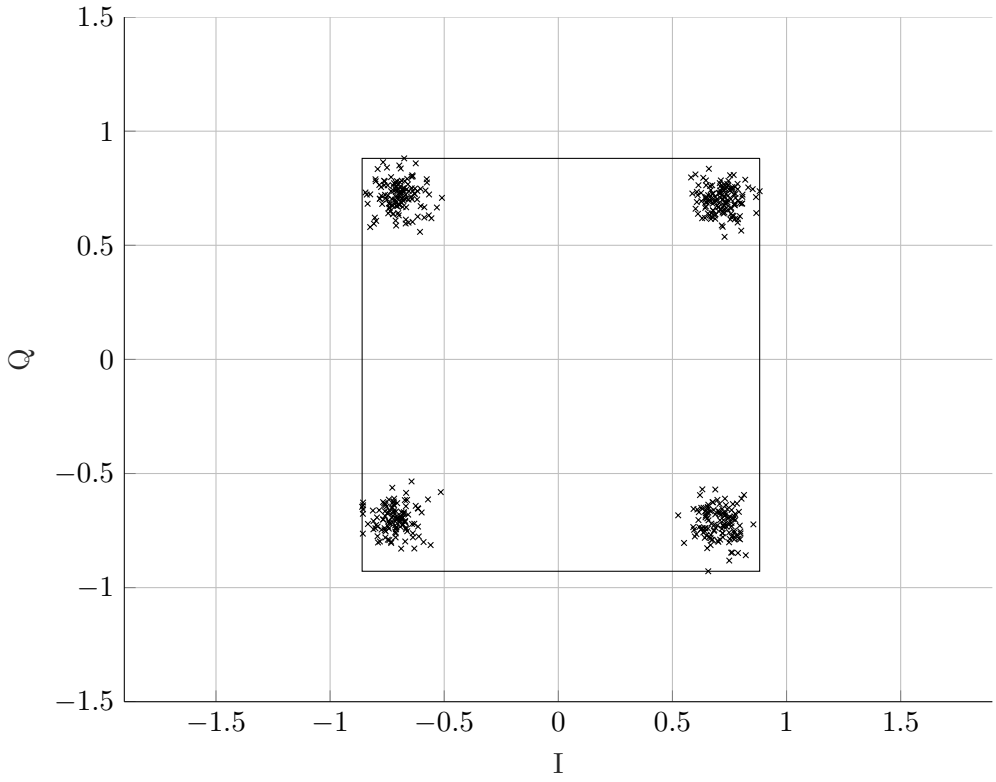


Fig. 3.4: Box fitted around rotated symbols.

the projected symbols is found and saved. This is the "max" value. After completing the test phases, the "max" values are compared. The rotation which yields the smallest "max" value – or the "minimum of the maxes" – is taken to be the best rotation to correct the phase offset.

3.4 Carrier frequency offset correction by spectral estimation

The spectrum of the fourth power of the matched filtered, timing-corrected signal contains a peak at the frequency bin νT [57], where ν is the frequency offset and T is the symbol period. The system assumes that the frequency offset is within $\pm 5\%$ of the demixing frequency. This limits the search to frequencies within $\pm .05/T$. An FFT is used to estimate the spectrum with an FFT length that is the smallest power of 2 larger than the length of the filtered signal. The actual frequency offset is calculated by

$$\nu = \arg\max(|\mathcal{F}\{y^4\}|) \cdot N/(4 \cdot nfft) \tag{3.5}$$

where y is the matched filtered signal, N is the upsampling factor, and $nfft$ is the length of the FFT. The offset is corrected by multiplying y by $e^{-j2\pi\nu T/N}$. Figure 3.5 shows the spectrum of the fourth power of the signal before and after carrier frequency offset correction.

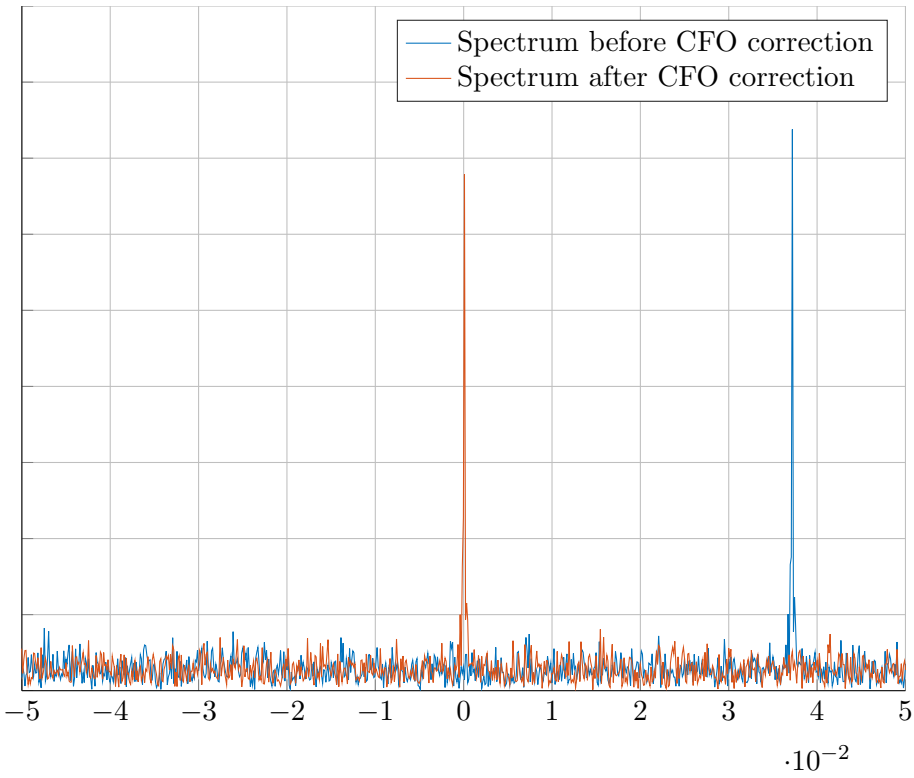


Fig. 3.5: Spectrum of $(y[n])^4$

3.5 System verification

The system was verified in the following ways:

- The proposed algorithm was first simulated in MATLAB, then implemented in C, then implemented on the GPU. Timing, phase, and frequency offset correction were verified separately and jointly.
- The noise level was verified by running a noiseless signal through the system, running a purely noisy signal through the system, and comparing the variances of the system outputs. This experiment yielded measurements very close to the specified SNR.
- Many iterations of the demodulator function were run to produce a bit error rate plot.

A complete system diagram is shown in Appendix B.

CHAPTER 4

CUDA IMPLEMENTATION AND METHODS

4.1 GPU Architecture

Graphics processing units (GPUs) can provide significant speedups over sequential CPU computing if the algorithm can be parallelized. The parallel behavior is achieved using a Single Instruction Multiple Data (SIMD) behavior. The work is divided such that a single instruction is executed by many processing cores, and each operates on different data. CUDA provides a useful abstraction for dividing up work into "threads," the individual units of work, and "blocks," or groups of threads with some shared resources. The exact configuration of threads and blocks is determined by the user at the launch of the kernel. Figure 4.1 [62] shows CUDA's memory hierarchy. Each thread has access to its own local and register memory, shared memory can be accessed by any thread in a block, and global memory can be accessed by any thread in any block. Memory access is an important consideration in GPU programming. Memory copies between the host and the device are computationally expensive, so the best programming strategy is to put the data on the device and leave it there for as long as possible. It is worthwhile to use the device to perform operations which may be more efficient on the host, because copying back and forth is time consuming. Memory considerations also come into play within the kernels themselves. Local and shared memory access is relatively fast, while global memory access tends to be slow. It is therefore ideal to reduce global memory reads whenever possible and to use shared memory instead.

4.2 Indexing

Blocks and grids are numbered as shown in Figure 4.1. Within each kernel, CUDA provides built-in variables for the block and grid dimensions, as well as the location of each

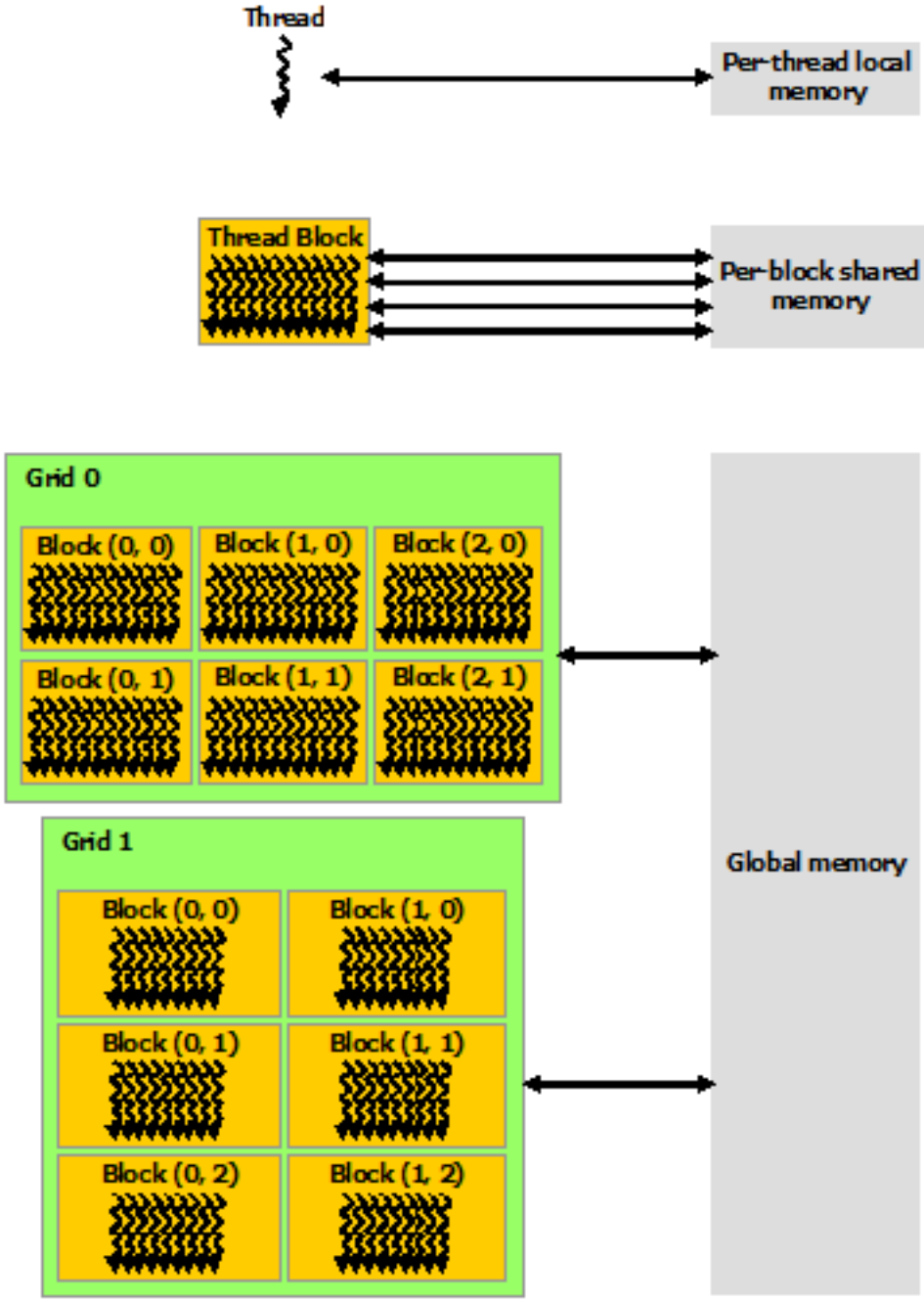


Fig. 4.1: CUDA Memory hierarchy

thread within the block and grid.

gridDim	Blocks per grid
blockIdx	Index of the block
blockDim	Threads per block
threadIdx	Index of the thread

This indexing is the mechanism by which each thread can do separate, independent work. The indexing is usually calculated as follows:

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
```

The thread can then use this "thread index" to read and write data at locations unique from any other thread.

The block size and grid size are defined by the user when the kernel is invoked. The number of threads per block is typically a multiple of 32, and the number of blocks per grid is usually defined as follows [62]:

```
int blksPerGrid = (N + threadsPerBlk - 1) / threadsPerBlk;
```

where N is the length of the array.

4.3 Parallel Reduction

Reduction algorithms are often useful in parallel computing [63]. In this project, reduction is used to find the sum, the minimum, or the maximum of an array. The reduction algorithm is implemented by launching a kernel and allowing the blocks to store initial results in shared memory. Then half of the threads remain idle while the other half computes results for the values in shared memory. This method is repeated until there is one final result per block, and a single thread writes its result to global memory. The diagram in Figure 4.2 demonstrates this process.

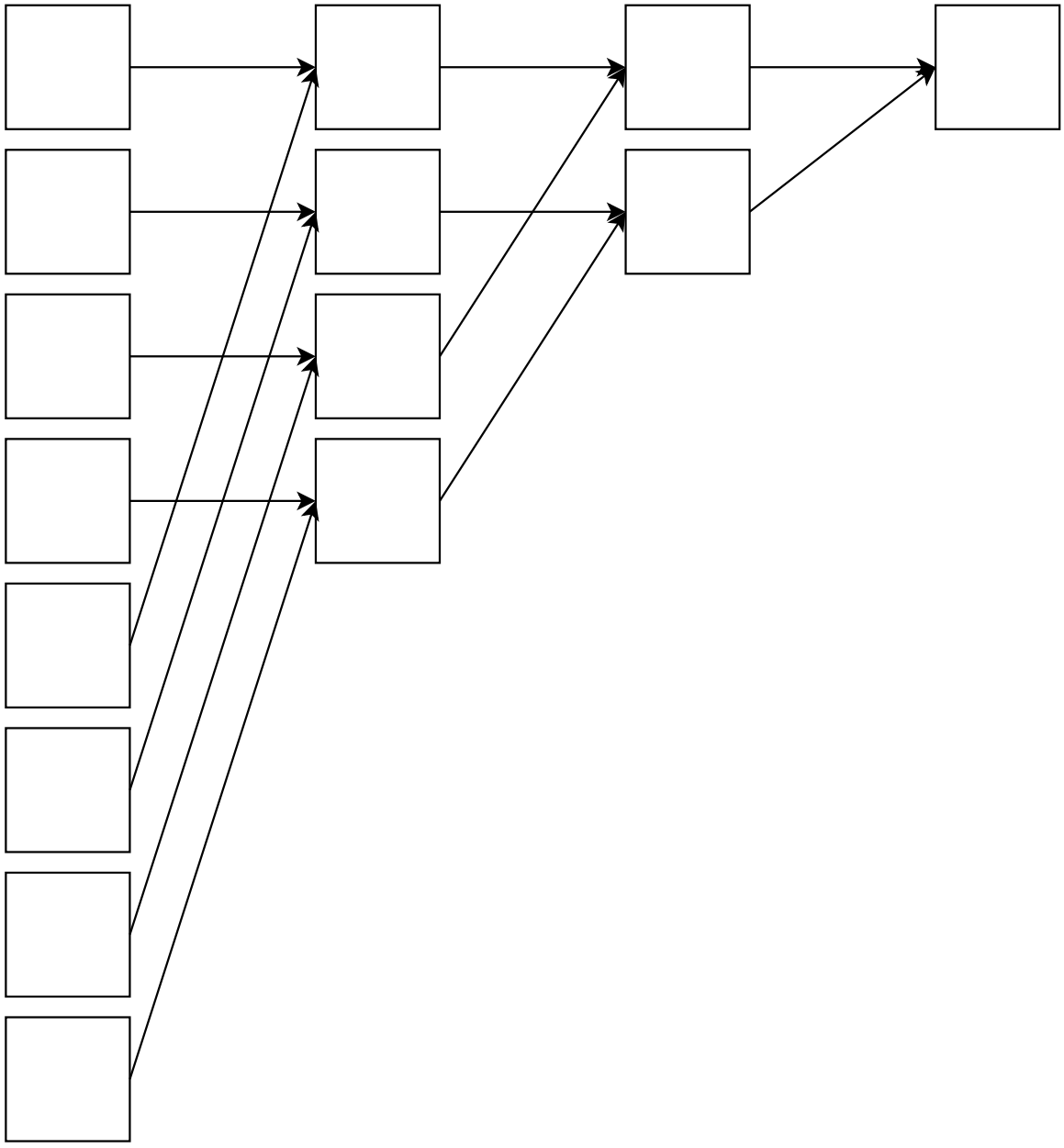


Fig. 4.2: Parallel reduction for a single block

The result is an array in global memory with the intermediate results from each block. Reduction could be performed on this final intermediate array using a single block, or the memory could be copied to the CPU and the final result could be found sequentially. The following example demonstrates the difference between sequential programming and parallel reduction.

Finding the maximum sequentially:

```
float max = -1e10; int ind = -1;
for(int i=0; i<len; i++){
    if(data[i] > max){
        max = data[i];
        ind = i;
    }
 }
```

Finding the maximum using parallel reduction:

1. The kernel is launched. Each thread computes a preliminary "local" result and stores it in shared memory.

```
int tid = blockIdx.x * blockDim.x + threadIdx.x;
int cacheIdx = threadIdx.x;    //loc. of thread in block
float max = -1e10; int ind = -1;
while(tid < len){
  if(data[tid] > max){
      max = data[tid];          //current maximum
      ind = tid;                //location of current maximum
  }
  tid += step;
}
maxCache[cacheIdx] = max;
indCache[cacheIdx] = ind;
__syncthreads();
```

2. Each block now has a shared array filled with local results. Half of the threads in each block perform the next computation and store their results back into the shared memory, while the other half of the threads remain idle. Thus, the problem is reduced

by half in each block. This step is repeated until there is only one active thread in the block.

```
int i = blockDim.x/2;
while(i != 0){
    if(cacheIdx < i){
    //bottom 1/2 of the threads perform the reduction
        if(maxCache[cacheIdx + i] > maxCache[cacheIdx]){
          maxCache[cacheIdx] = maxCache[cacheIdx + i];
          indCache[cacheIdx] = indCache[cacheIdx + i];
        }
    }
    __syncthreads();
    i /= 2;
}
__syncthreads();
```

3. A single thread holds the intermediate result for its block. This thread writes its result to global memory.

```
if(cacheIdx == 0){
    maxArr[blockIdx.x] = maxCache[0];
    indArr[blockIdx.x] = indCache[0];
}
```

4. The return array holds *threadsPerBlock* intermediate results. The final result can be found by looping through the data on the CPU, or by running the reduction function again with only one block.

4.4 Methods of convolution

This project explored various methods of performing filtering in parallel. Three methods are compared: traditional or "inner product" convolution, multi-threaded convolution [2], and fast convolution using cuFFT [64]. The convolution methods were timed using arbitrary test data and a 121-point filter. All experiments were executed on an Nvidia GeForce GTX 1070 graphics card.

4.4.1 Inner-product Convolution

Equation (4.1) represents linear convolution of an FIR filter.

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] \tag{4.1}$$

The "inner product" convolution method is performed in parallel with each thread computing a single output and looping through all filter values. It is based on a view of the output as a sum of inner-products between the input samples and the filter. In other words, each thread computes $y[n]$ from (4.1), where n is determined by the thread index. The GPU execution times of the inner-product convolution are shown in Table 4.1.

Block size	Grid size	Min time [μs]	Mean time [μs]	Max time [μs]
32	82	11.136	11.319	12.256
64	41	10.849	11.094	11.904
128	21	10.656	10.891	11.968
256	11	11.744	11.902	12.832
512	6	12.288	12.827	13.921
1024	3	18.848	18.942	19.488

Table 4.1: Run times for inner-product convolution method

4.4.2 Multi-threaded convolution

The "multi-threaded" convolution method is based on a view of the output points as a linear combination of the filter values. As each input sample comes in, it is multiplied by

each of the filter points and accumulated separately. The input samples do not need to be stored in memory, as each accumulator uses each input sample simultaneously and then it is no longer needed. When the accumulator has an output point ready, it writes its result to the output array and resets. Each accumulator in a block takes the same input value and multiplies it by a different filter value. Each block overlaps by N, where N is the length of the filter, to prevent a transient response at the beginning of each block. A snippet of the code is shown below.

```
int l = threadIdx.x;      //accumulator idx
int i = 0;                 //current time idx
while(tid < end){
  m = (l-i+N)%N;
  //circular index: current filter idx for this thread
  if(tid < dataLen && tid >=str) //if tid is in the valid range
    x = dataArr[tid];     //read new input
  else
    x = 0.0;
  acc += filt[m]*x;       //multiply and accumulate
  if(m == 0){             //result is ready
    if(tid>= str && tid < sigLen){
      convArr[tid] = acc;//write to output array
      acc = 0.0;          //reset accumulator
    }
  }
  i = (i+1+N)%N;
  tid++;
}
```

The algorithm depends on having a block size greater than the length of the filter. In this case, the filter is 121 points long, so blocks with fewer than 128 threads result in errors.

The timing results for the multi-threaded method are shown in Table 4.2.

Block size	Grid size	Min time [μs]	Mean time [μs]	Max time [μs]
128	21	42.464	42.702	43.872
256	11	63.585	63.798	64.865
512	6	116.74	116.92	117.73
1024	3	323.68	324.11	325.00

Table 4.2: Run times for multi-threaded convolution method

4.4.3 Fast Convolution

Fast convolution is implemented using the cuFFT API. The program takes the FFT of both the filter and the data array, multiplies them element-wise, and takes the inverse FFT. Timing results for fast convolution are shown in Table 4.3.

Block size	Grid size	Min time [ms]	Mean time [ms]	Max time [ms]
32	82	26.9	31.6	56.5
64	41	27.2	30.2	41.4
128	21	27.0	31.9	46.8
256	11	26.7	30.9	52.9
512	6	26.9	30.9	58.9
1024	3	27.1	31.9	46.2

Table 4.3: Run times for fast convolution method

4.4.4 Conclusions

From the results in the previous sections, it appears that the inner-product form of convolution generally runs the fastest, with an average execution time of 10.891 μs for a block and grid size of 128 and 21, respectively. The multi-threaded convolution execution time for the same block and grid size is 42.702 μs on average. The fast convolution method

execution speeds are roughly constant across block and grid size.

CHAPTER 5

RESULTS

All tests were performed using an Nvidia GeForce GTX 1070 graphics card.

5.1 Optimal synchronization parameters

By nature of the approach used to perform the timing and phase correction, the performance may be affected by the specific parameters chosen to run the algorithm. In the timing correction case, the system generates a bank of matched filters and selects the filter which results in the best timing correction. In running the simulation, it was necessary to determine how many matched filters were needed to effectively resolve the sample timing offset without slowing down the system. The more filters used, the better on average the system will be able to find the correct timing offset. In order to determine the appropriate number of matched filters for the receiver, bit error rate tests were run for several different values and compared. The plot in Figure 5.1 shows the results of these tests.

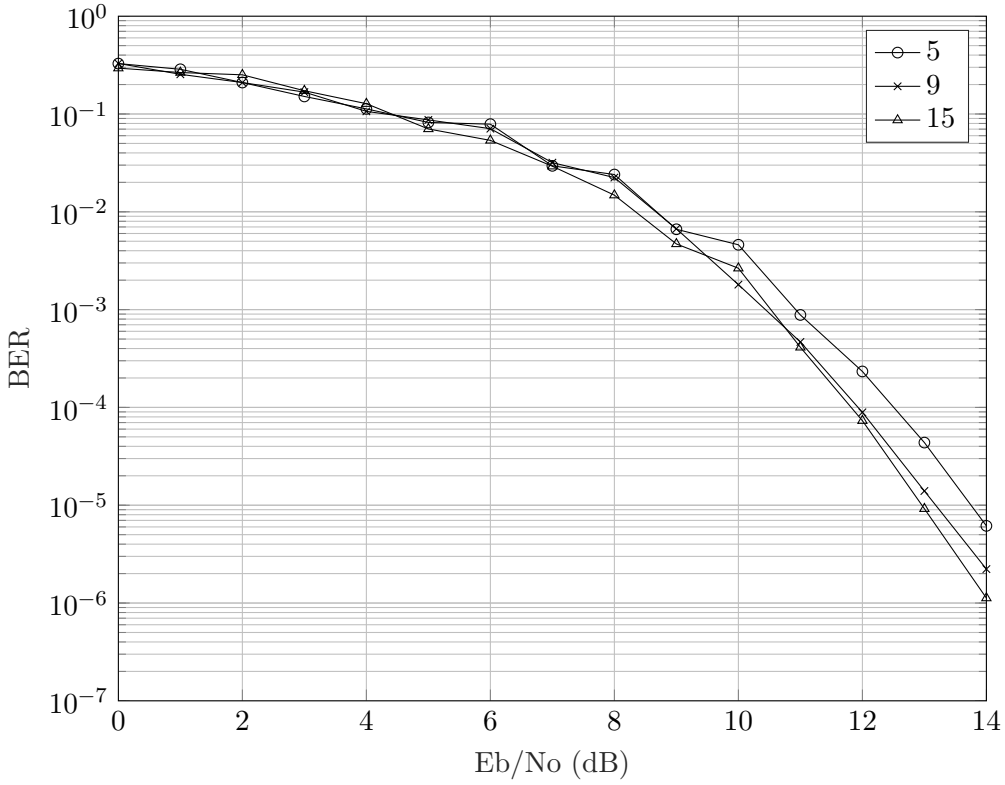


Fig. 5.1: Receiver BER by number of matched filters

A large number of matched filters used does not seem to make a substantial improvement to the performance. However, additional filters require additional computation, so it is desirable to choose the minimum number of filters that give an acceptable performance. Five matched filters were selected as the default value.

Similarly, in the phase correction case, multiple different phase rotations are tested and the best is selected. The number of test phases corresponds with the ability of the receiver to resolve the phase. The results of the bit error rate tests are shown in Figure 5.2. In the figure, 'np' indicates the number of test phases used in the phase correction.

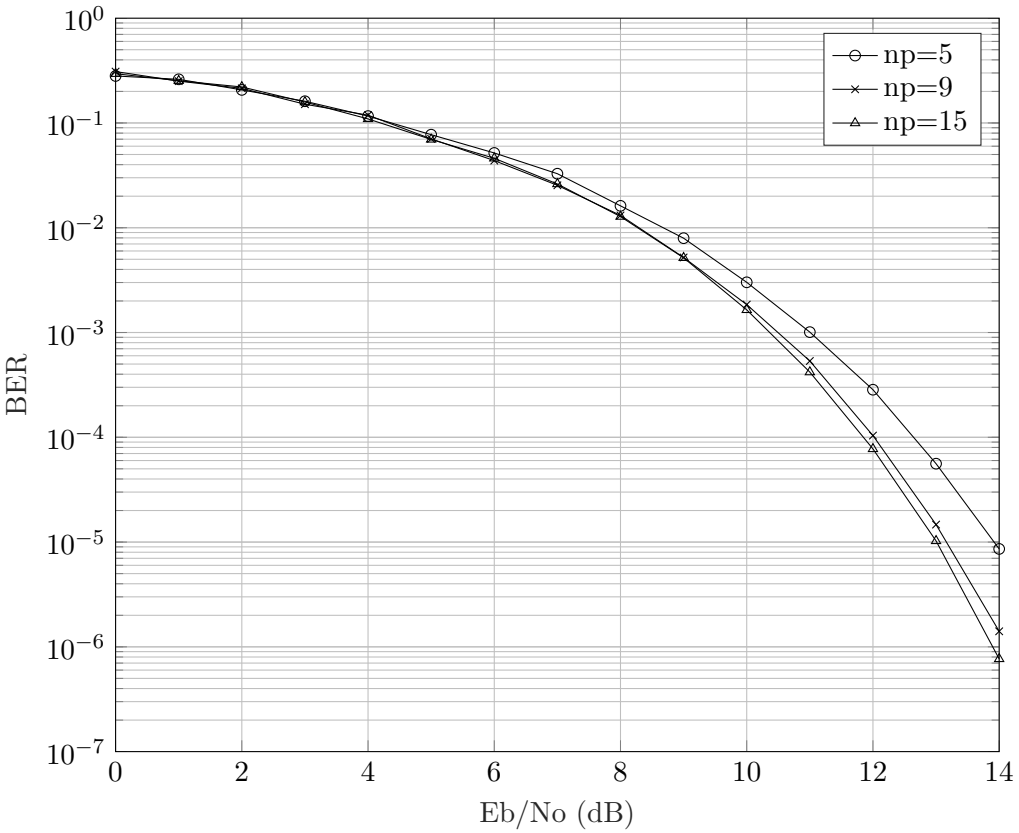


Fig. 5.2: Receiver BER by number of phase points tested.

Unlike the timing synchronization subsystem, the phase correction subsystem does improve with more test points.

5.2 Comparison of phase synchronization methods

Two methods were implemented for phase offset correction: the kurtosis method and the "min/max" method, both described in Chapter 3. The min/max method is an ad hoc method developed for this project. The methods are compared based on their accuracy and their runtime. Figure 5.3 compares the two phase correction methods based on their accuracy. The results indicate that the kurtosis method does a much better job of correcting phase offset in terms of bit error rate, even with fewer phase test points.

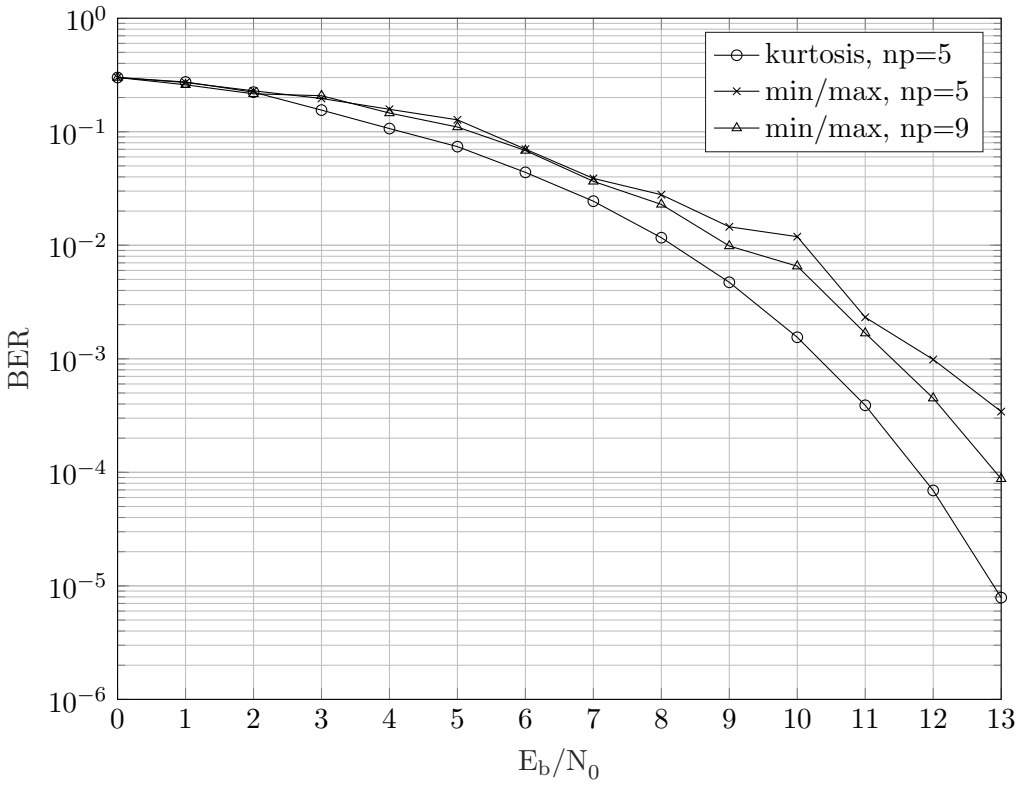


Fig. 5.3: Receiver BER by method of phase correction

Another important consideration in the choice of method is the time it takes to run. Table 5.1 shows the results of repeated GPU timing tests with each phase correction method.

Method	Min time [ms]	Avg time [ms]	Max time [ms]
Kurtosis	0.2370	0.8073	0.9303
Min/max	0.1102	0.4136	0.4745

Table 5.1: GPU execution time for phase correction with 100 tests

The min/max method runs significantly faster than the kurtosis method on average due to the higher complexity computations used to compute the kurtosis.

Based on these results, it can be concluded that the kurtosis method is a more robust method of phase correction, but has a higher level of complexity. Since the min/max method

relies on the most extreme (or "noisiest") received point, it makes sense that its performance is heavily affected by noise.

On the other hand, the kurtosis method requires computing the expected value of the second and fourth powers of the data sequence, as in Equation (3.4). Power operations are computationally expensive, which accounts for the longer runtime of the kurtosis-based phase correction.

5.3 CUDA Optimizations

The purpose of this implementation is to demonstrate the viability of GPU processing in performing receiver-side processing in a communications system. That viability is largely measured in terms of speed and data rate. Thus, it is beneficial to consider what measures can be taken to minimize the runtime of the demodulator and thereby increase the data rate. A few of these measures are considered here.

5.3.1 Symbol block size

The best symbol packet length was investigated in terms of runtime and performance. In Table 5.2, the demodulator runtime is compared for different symbol packet lengths.

# of symbols	Time/packet	Time/symbol
10	0.838790	0.083879
25	0.955798	0.038232
50	1.148694	0.022974
100	1.520406	0.015204
250	2.681757	0.010727
500	4.633450	0.009267
1000	8.337942	0.008338

Table 5.2: Average runtime in milliseconds by packet length.

In general, it was found that the longer symbol packet lengths had lower processing times per symbol and therefore higher data rates. However, they also tended to have some minor increases in error rates. For most experiments, a packet length of 500 symbols was used.

Table 5.3 compares the data rates of the demodulator based on the number of threads per block. The number of blocks per grid is calculated based on the block size as shown in 4.2.

Threads/blk	Blks/grid	Min	Mean	Max
32	32	1.5764	1.6350	1.6875
256	4	1.3712	1.6081	1.6638

Table 5.3: Data rates in Mbps based on block and grid size.

5.4 Timing tests

A general breakdown of the runtime of each function in the demodulator is shown in Table 5.4. This experiment was performed with 500 symbols per packet and 64 threads per block.

Code snippet	GPU time (ms)
Entire demodulation	.720096
Timing corr. block	.291920
Convolution (inner product)	.019817
Downsampling	.003072
Complex Kurtosis	.015360
Find min location	.005120
CFO correction block	.102400
FFT execution	.037313
Compute cmplx mag.	.003152
Finding max location	.048147
Phase offs. corr. block	.257184
Rotation	.004678
Real Kurtosis	.052266

Table 5.4: Code Timing Tests

The complex kurtosis calculation and the filtering were consistently the most computationally expensive functions in the receiver. Alternative convolution approaches were explored in 4.4.

Table 5.5 shows the results of timing tests on the two different demodulation implementations. The C code was compiled with standard GCC optimization flags.

Each implementation was run 100 times and the minimum, maximum, and average runtimes are presented in the table.

Impl.	Min Time [ms]	Mean Time [ms]	Max Time [ms]
C	10.9630	12.3832	32.6510
CUDA	0.5580	0.5832	0.6260

Table 5.5: Demodulation runtime for 500-symbol packets of data.

The CUDA implementation offers speedups of around an average of 20 times over the C implementation. The average data rate for the C implementation is 80 kbps and the average data rate for the CUDA implementation is 1.7 Mbps.

CHAPTER 6

CONCLUSION

In Chapter 2, the relevant literature was examined, and a few trends were noted.

First, there is always a need for faster and more reliable communications systems. A growing demand for data drives the search for techniques that enable higher and higher data rates, while still balancing the need for high performance and low computational complexity. While continuous data stream transmission is still used, there are plenty of applications more suited to packetized transmission. Synchronization in packet-based communications system requires different approaches compared to the traditional PLL-based methods. Commonly, packets are sent with known symbols – pilot data – from which the receiver is able to learn what timing and phase corrections to apply to an incoming signal. However, this data-aided approach wastes valuable bandwidth by sending known symbols with every packet. To increase data rates, blind synchronization techniques are utilized to perform timing and phase recovery, but these techniques often come at the cost of lower performance.

Second, the space industry in particular is looking to transmit data at high rates as small satellite missions increase. Satellites may be gathering large amounts of scientific data and may only have brief contact with ground stations, thus driving a need for fast transmission. With high transmission rates, fast demodulation is desired on the ground station side to process the data in real time.

Third, software defined radio is an attractive option for satellites and ground stations alike. In the past few decades the idea of the software defined radio has become much more realistic as processors have become faster, cheaper, and more lightweight. Software radios are beneficial for their flexibility, easy maintenance, replicability, and reasonably low cost. Furthermore, they are well suited to changing protocols and complex modulation schemes. Their development time is low, as updates can be pushed to multiple units in different locations simultaneously. In a well designed system there is no need to replace any

hardware to accommodate a different protocol.

Finally, GPUs have become an attractive option for high speed computing in a wide variety of disciplines. No longer used only for graphical rendering, GPUs are seeing considerable growth in scientific fields.

All these factors are combined together in this paper to present a GPU-based demodulation scheme for a ground station software defined radio. The algorithm presented in Chapter 3 and introduced in [57] is by no means optimal, but it is fairly low complexity and can be implemented in a parallel environment. Some of the techniques used for doing so are discussed in Chapter 4. Results of experiments performed with the system are described in Chapter 5.

These results demonstrate that not only is the GPU a viable platform for communications system functions, but their potential has yet to be fully explored. A comparison of the algorithm run on a CPU versus a GPU shows a notable improvement in data rate simply by adapting the algorithm to the GPU. One of the compelling features of a GPU is how simple it is to increase the speed. If, as was shown here, it is possible to demodulate 2 Mbps on a single data stream, then that data rate can be easily increased by simply boosting the hardware resources and running the program on multiple data streams in parallel. The data rate is increased without any significant change to the algorithm.

What's more, the system presented here makes no claim of optimality, and yet still achieves substantial speedups. It is not difficult to imagine that with more work and optimization, the GPU could be a powerful choice for a software defined radio.

6.1 Future work

The potential for GPUs in communications applications is rich and diverse. The system implemented in this paper assumes an Additive White Gaussian Noise channel. In the case of a multi-path or frequency selective fading channel, some equalization algorithm would need to be applied. A study could be made on the efficacy of such an approach to equalization. Further, there has already been some work on error correction codes on

GPUs [53–56]. More work could be done on implementing such codes and optimizing them for the GPU setting.

REFERENCES

[1] C. S. Fish, C. M. Swenson, G. Crowley, A. Barjatya, T. Neilsen, J. Gunther, I. Azeem, M. Pilinski, R. Wilder, D. Allen, M. Anderson, B. Bingham, K. Bradford, S. Burr, R. Burt, B. Byers, J. Cook, K. Davis, C. Frazier, S. Grover, G. Hansen, S. Jensen, R. LeBaron, J. Martineau, J. Miller, J. Nelsen, W. Nelson, P. Patterson, E. Stromberg, J. Tran, S. Wassom, C. Weston, M. Whiteley, Q. Young, J. Petersen, S. Schaire, C. R. Davis, M. Bokaie, R. Fullmer, R. Baktur, J. Sojka, and M. Cousins, "Design, development, implementation, and on-orbit performance of the dynamic ionosphere cubesat experiment mission," *Space Science Reviews*, vol. 181, no. 1, pp. 61–120, May 2014. [Online]. Available: https://doi.org/10.1007/s11214-014-0034-x

[2] J. Gunther, H. Gunther, and T. Moon, "GPU acceleration of DSP for communication receivers," *Proceedings of the GNU Radio Conference*, vol. 2, no. 1, p. 9, 2017. [Online]. Available: https://pubs.gnuradio.org/index.php/grcon/article/view/39

[3] B. Klofas, "Cubesat communication systems: 2003 - 2018," https://www.klofas.com/comm-table/, Nov 2018.

[4] M. Rice, *Digital Communications: A Discrete-time Approach*. Pearson/Prentice Hall, 2009. [Online]. Available: https://books.google.com/books?id=EB3r7JtXlWwC

[5] F. Gardner, "A BPSK/QPSK timing-error detector for sampled receivers," *IEEE Transactions on Communications*, vol. 34, no. 5, pp. 423–429, May 1986.

[6] U. Mengali, *Synchronization Techniques for Digital Receivers*, ser. Applications of Communications Theory. Springer US, 1997. [Online]. Available: https://books.google.com/books?id=89Gscsw7PvoC

[7] F. Gardner, *Demodulator Reference Recovery Techniques Suited for Digital Implementation*. Gardner Research Comp., 1988. [Online]. Available: https://books.google.com/books?id=BqQ_NQAACAAJ

[8] K. Mueller and M. Muller, "Timing recovery in digital synchronous data receivers," *IEEE Transactions on Communications*, vol. 24, no. 5, pp. 516–531, May 1976.

[9] C. W. Farrow, "A continuously variable digital delay element," in *1988., IEEE International Symposium on Circuits and Systems*, June 1988, pp. 2641–2645 vol.3.

[10] D. Fu and A. N. Willson, "Trigonometric polynomial interpolation for timing recovery," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 52, no. 2, pp. 338–349, Feb 2005.

[11] D. Kim, M. J. Narasimba, and D. C. Coc, "Design of optimal interpolation filter for symbol timing recovery," *IEEE Transactions on Communications*, vol. 45, no. 7, pp. 877–884, July 1997.

[12] f. j. harris and M. Rice, "Multirate digital filters for symbol timing synchronization in software defined radios," *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 12, pp. 2346–2357, Dec 2001.

[13] X. Ma, H. Zhao, G. Li, and Y. Zhao, "Implementation of a high-throughput OFDM system using graphics processing units," in *2013 15th IEEE International Conference on Communication Technology*, Nov 2013, pp. 639–644.

[14] Q. Zhao and G. L. Stuber, "Robust time and phase synchronization for continuous phase modulation," *IEEE Transactions on Communications*, vol. 54, no. 10, pp. 1857–1869, Oct 2006.

[15] E. Hosseini and E. Perrins, "Timing, carrier, and frame synchronization of burst-mode CPM," *IEEE Transactions on Communications*, vol. 61, no. 12, pp. 5125–5138, December 2013.

[16] Y.-L. Huang, K.-D. Fan, and C.-C. Huang, "A fully digital noncoherent and coherent GMSK receiver architecture with joint symbol timing error and frequency offset estimation," *IEEE Transactions on Vehicular Technology*, vol. 49, no. 3, pp. 863–874, May 2000.

[17] M. Oerder and H. Meyr, "Digital filter and square timing recovery," *IEEE Transactions on Communications*, vol. 36, no. 5, pp. 605–612, May 1988.

[18] M. Morelli, A. N. D'Andrea, and U. Mengali, "Feedforward ML-based timing estimation with PSK signals," *IEEE Communications Letters*, vol. 1, no. 3, pp. 80–82, May 1997.

[19] E. Panayirci and E. K. Bar-Ness, "A new approach for evaluating the performance of a symbol timing recovery system employing a general type of nonlinearity," *IEEE Transactions on Communications*, vol. 44, no. 1, pp. 29–33, Jan 1996.

[20] S. J. Lee, "A new non-data-aided feedforward symbol timing estimator using two samples per symbol," *IEEE Communications Letters*, vol. 6, no. 5, pp. 205–207, May 2002.

[21] M. Marey, O. A. Dobre, and B. Liao, "Second-order statistics-based blind synchronization algorithm for two receive-antenna orthogonal STBC systems," *IEEE Communications Letters*, vol. 18, no. 7, pp. 1115–1118, July 2014.

[22] B. Park, H. Cheon, E. Ko, C. Kang, and D. Hong, "A blind OFDM synchronization algorithm based on cyclic correlation," *IEEE Signal Processing Letters*, vol. 11, no. 2, pp. 83–85, Feb 2004.

[23] M. Moeneclaey and T. Batsele, "Carrier-independent nda symbol synchronization for m-psk, operating at only one sample per symbol," in *[Proceedings] GLOBECOM '90: IEEE Global Telecommunications Conference and Exhibition*, Dec 1990, pp. 594–598 vol.1.

[24] M. Flohberger, W. Gappmair, and S. Cioni, "Two iterative algorithms for blind symbol timing estimation of m-psk signals," in *2009 International Workshop on Satellite and Space Communications*, Sep. 2009, pp. 8–12.

[25] F. Xiong and M. Andro, "The effect of doppler frequency shift, frequency offset of the local oscillators, and phase noise on the performance of coherent OFDM receivers," NASA, Tech. Rep., Apr 2001.

[26] L. Cimini, "Analysis and simulation of a digital mobile channel using orthogonal frequency division multiplexing," *IEEE Transactions on Communications*, vol. 33, no. 7, pp. 665–675, Jul 1985.

[27] M. Morelli and U. Mengali, "Feedforward frequency estimation for psk: A tutorial review," *European Transactions on Telecommunications*, vol. 9, no. 2, pp. 103–116, 1998. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.4460090203

[28] D. Rife and R. Boorstyn, "Single tone parameter estimation from discrete-time observations," *IEEE Transactions on Information Theory*, vol. 20, no. 5, pp. 591–598, Sep. 1974.

[29] M. Luise and R. Reggiannini, "Carrier frequency recovery in all-digital modems for burst-mode transmissions," *IEEE Transactions on Communications*, vol. 43, no. 2/3/4, pp. 1169–1178, Feb 1995.

[30] S. Tretter, "Estimating the frequency of a noisy sinusoid by linear regression (corresp.)," *IEEE Transactions on Information Theory*, vol. 31, no. 6, pp. 832–835, November 1985.

[31] S. Kay, "A fast and accurate single frequency estimator," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 12, pp. 1987–1990, Dec 1989.

[32] M. Morelli and U. Mengali, "Carrier-frequency estimation for transmissions over selective channels," *IEEE Transactions on Communications*, vol. 48, no. 9, pp. 1580–1589, Sep. 2000.

[33] A. Yilmaz, M. Kesal, and F. A. Onat, "Frequency estimation for burst communication based on irregular repetition of data symbols," in *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*, Oct 2018, pp. 1–9.

[34] C. Bosquez, R. Moreira, and A. De La Cruz, "Alert system for emergency vehicles using software-defined radio," in *2017 IEEE International Conference on Microwaves, Antennas, Communications and Electronic Systems (COMCAS)*, Nov 2017, pp. 1–5.

[35] I. Vitas, D. imuni, and P. Kneevi, "Evaluation of software defined radio systems for smart home environments," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2015, pp. 562–565.

[36] Y. Pei, H. Chen, and B. Pei, "Implementation of GPS software receiver based on GNU radio," in *2018 Cross Strait Quad-Regional Radio Science and Wireless Technology Conference (CSQRWC)*, July 2018, pp. 1–3.

[37] N. Hosseini and D. W. Matolak, "Software defined radios as cognitive relays for satellite ground stations incurring terrestrial interference," in *2017 Cognitive Communications for Aerospace Applications Workshop (CCAA)*, June 2017, pp. 1–4.

[38] A. C. Clarke, "Extra-terrestrial relays: Can rocket stations give world-wide radio coverage?" in *Communication Satellite Systems Technology*, ser. Progress in Astronautics and Rocketry, R. B. Marsten, Ed. Elsevier, 1966, vol. 19, pp. 3 – 6. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9781483227160500062

[39] R. E. Sheriff and A. R. L. Tatnall, *Telecommunications*. John Wiley & Sons, Ltd, 2011, ch. 12, pp. 395–437. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119971009.ch12

[40] M. R. Maheshwarappa and C. P. Bridges, "Software defined radios for small satellites," in *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, July 2014, pp. 172–179.

[41] M. R. Maheshwarappa, M. Bowyer, and C. P. Bridges, "Software defined radio (SDR) architecture to support multi-satellite communications," in *2015 IEEE Aerospace Conference*, March 2015, pp. 1–10.

[42] P. Angeletti, M. Lisi, and P. Tognolatti, "Software defined radio: A key technology for flexibility and reconfigurability in space applications," in *2014 IEEE Metrology for Aerospace (MetroAeroSpace)*, May 2014, pp. 399–403.

[43] T. Hobiger, R. Haas, and J. Strandberg, "Ground-based GNSS-R solutions by means of software defined radio," in *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, July 2016, pp. 5635–5637.

[44] V. Dascal, P. Dolea, O. Cristea, and T. Palade, "Low-cost SDR-based ground receiving station for LEO satellite operations," in *2013 11th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services (TELSIKS)*, vol. 02, Oct 2013, pp. 627–630.

[45] J.-J. M. Jyh-Ching Juang, Chiu-Teng Tsai, *Small Satellites for Earth Observation*. Springer, Dordrecht, 2008, ch. A Software-Defined Radio Approach for the Implementation of Ground Station Receivers.

[46] B. Paillassa and C. Morlet, "Flexible satellites: software radio in the sky," in *10th International Conference on Telecommunications, 2003. ICT 2003.*, vol. 2, Feb 2003, pp. 1596–1600 vol.2.

[47] F. Iacomacci, C. Morlet, F. Autelitano, G. C. Cardarilli, M. Re, E. Petrongari, G. Bogo, and M. Franceschelli, "A software defined radio architecture for a regenerative onboard processor," in *2008 NASA/ESA Conference on Adaptive Hardware and Systems*, June 2008, pp. 164–171.

[48] X. Cai, M. Zhou, and X. Huang, "Model-based design for software defined radio on an FPGA," *IEEE Access*, vol. 5, pp. 8276–8283, 2017.

[49] L. Xu, N. I. Ziedan, X. Niu, and W. Guo, "Correlation acceleration in GNSS software receivers using a CUDA-enabled GPU," *GPS Solutions*, vol. 21, no. 1, pp. 225–236, Jan 2017. [Online]. Available: https://doi.org/10.1007/s10291-016-0516-2

[50] R. Muzammil, M. S. Beg, M. M. Jamali, and M. W. Majid, "Design and testing of a software defined radio based transceiver on a graphics processing unit," in *2012 Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, Nov 2012, pp. 1107–1110.

[51] M. Lai and T. Chiueh, "Implementation of a C-V2X receiver on an over-the-air software-defined-radio platform with OpenCL," in *2018 New Generation of CAS (NGCAS)*, Nov 2018, pp. 170–173.

[52] L. Stolz, M. Ihmig, and W. Stechele, "An evaluation on using GPU coprocessing for software radios on a low-cost platform," in *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*, Oct 2012, pp. 1–8.

[53] C. H. Chan and F. C. M. Lau, "Parallel decoding of LDPC convolutional codes using OpenMP and GPU," in *2012 IEEE Symposium on Computers and Communications (ISCC)*, July 2012, pp. 000 225–000 227.

[54] S. Keskin and T. Kocak, "GPU-based gigabit LDPC decoder," *IEEE Communications Letters*, vol. 21, no. 8, pp. 1703–1706, Aug 2017.

[55] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," in *2013 IEEE Global Conference on Signal and Information Processing*, Dec 2013, pp. 1258–1261.

[56] C. Lin, W. Liu, W. Yeh, L. Chang, W. W. Hwu, S. Chen, and P. Hsiung, "A tiling-scheme viterbi decoder in software defined radio for GPUs," in *2011 7th International Conference on Wireless Communications, Networking and Mobile Computing*, Sep. 2011, pp. 1–4.

[57] J. Gunther and T. Moon, "Burst mode synchronization of QPSK on AWGN channels using kurtosis," *IEEE Transactions on Communications*, vol. 57, no. 8, pp. 2453–2462, Aug 2009.

[58] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Ann. Math. Statist.*, vol. 29, no. 2, pp. 610–611, 06 1958. [Online]. Available: https://doi.org/10.1214/aoms/1177706645

[59] A. Hyvarinen, J. Karhunen, and E. Oja, *Independent Component Analysis*. John Wiley & Sons, 2001.

[60] J. J. Shynk, *Probability, random variables, and random processes: theory and signal processing applications*. Wiley-Blackwell, 2013.

[61] Z. Ding and Y. Li, *Blind equalization and identification*. Marcel Dekker, 2001.

[62] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2018, version 9.2.

[63] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley/Pearson Education, 2011.

[64] NVIDIA Corporation, "cuFFT Library User's Guide," 2018, version 9.2.

APPENDICES

APPENDIX A

CODE LISTINGS

A.1 C Functions

functions.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <time.h>

#define PI 3.14159265358979

typedef struct{
    float re;
    float im;
}complx;

void diffenc(int *inbits, int *delta, int nbits);
//Performs differential encoding
//inbits: pointer to an array of input bits
//delta : pointer to output array of differentially encoded bits
//nbits : number of bits in the arrays

void diffdec(int *outbits, int *delta, int nbits);
//Performs differential decoding
//outbits: pointer to output array of bits
//delta  : pointer to input array of demodulated bits
//nbits  : number of bits in the arrays

void bits2sym(complx *syms, int *bits, int nsym, int bps, complx *lut);
//Converts an array of bits into complex QPSK symbols. realistically, this is only correct for qpsk
//syms  : complex array of output symbols (uses complx struct)
//bits  : input array of (differentially encoded) bits
//nsym  : number of symbols (should be number of bits / bits per symbol)
//bps   : bits per symbol (equals 2 for qpsk)
//lut   : complex array containing the lookup table for the constellation

void srrcDelay(float *p, float alpha, float N, int Lp, float Ts, float tau, int rev);
//Creates a square root raised cosine pulse shaping filter with a time delay
```

```
//p      : pointer to the output array (the pulse)
//alpha  : excess bandwidth parameter
//N      : upsampling factor
//Lp     : pulse truncation length
//Ts     : sample period
//tau    : time delay
//rev    : reversal flag -- if rev == 1, the output array is the flipped version

void conv(float *retArr, float *filter, float *dataArr, int filtLen, int dataLen);
//Performs linear convolution
//retArr: pointer to the return array
//filter: pointer to the input filter array
//dataArr: pointer to the input data array
//filtLen: length of the filter
//dataLen: length of the data array (# of data points)

void decisionBlk(int *bits, float *isyms, float *qsyms, complx *lut, int nsym, int bps);
//Outputs hard decisions on demodulated symbols (inverse of bits2sym)
//bits  : pointer to the output array of decided bits
//isyms : pointer to the array of the in-phase portion of the symbols (input)
//qsyms : pointer to the array of the quad-phase portion of the symbols (input)
//lut   : complex array containing the lookup table for the constellation
//nsym  : number of input symbols
//bps   : number of bits per symbol (but really, it only works for qpsk)

void boxmuller(float *noise, int len, float mean, float var);
//Generates gaussian distributed noise by the Box-Muller method
//noise: pointer to return array of noise data
//len:   length of noise array
//mean:  mean of the generated noise
//var:   variance of the generated noise

float dstCmplx(float srcReal, float srcImag, float dstReal, float dstImag);
//Calculates the distance between two complex numbers
//srcReal: real portion of the starting point
//srcImag: imaginary portion of the starting point
//dstReal: real portion of the ending point
//dstImag: imaginary portion of the ending point
//returns the distance (a float)

float absCmplx2(float numReal, float numImag);
//Calculates the squared magnitude of a complex number
//numReal: real portion of the input complex number
//numImag: imag portion of the input complex number
//returns the squared magnitude (a float)

float absCmplx(float numReal, float numImag);
//Calculates the magnitude of a complex number
//numReal: real portion of the input complex number
//numImag: imag portion of the input complex number
//returns the magnitude (a float)

float kurtCmplx(float *datReal, float *datImag, int len);
//Computes the complex kurtosis of an input sequence
//datReal: pointer to the real portion of the input data
```

```
//datImag: pointer to the imaginary portion of the input data
//len     : length of the input data
//returns the complex kurtosis (a float)

float kurtReal(float *data, int len);
//Computes the kurtosis for a zero-mean, real valued sequence
//data  : pointer to the input array of data
//len   : length of the input data

float deg2rad(float angle);
//Converts an angle in degrees to an angle in radians
//angle : input angle in degrees
//returns the angle in radians (a float)

float rad2deg(float phase);
//Converts an angle in radians to an angle in degrees
//phase : input angle in radians
//returns the angle in degrees (a float)

float getPower(float *data, int len);
//Computes the RMS power of an input sequence
//data  : pointer to the input data
//len   : length of the input data
//returns the RMS power (a float)

complx cmpSq(float numReal, float numImag);
//Computes the square of a complex number
//numReal : real portion of the input complex number
//numImag : imag portion of the input complex number
//returns a complex number equal to (numReal + j*numImag)^2 (a complx)

complx cmpAdd(complx num1, complx num2);
//Computes the sum of two complex numbers
//num1  : complex number to be added
//num2  : complex number to be added
//returns the complex sum of the two inputs (a complx)

float rand_float();
//Returns a random, uniformly distributed number between 0 and 1

#endif
```

functions.c

```
#include "functions.h"

void diffenc(int *inbits, int *delta, int nbits){
    int dprev[2]={0,0};
    int i;
    for(i=0; i<nbits-1; i+=2){
        if(inbits[i] == 0 && inbits[i+1] == 0){
            delta[i] = dprev[0];
            delta[i+1] = dprev[1];
        }else if(inbits[i] == 0 && inbits[i+1] == 1){
            delta[i] = dprev[1];
            delta[i+1] = 1-dprev[0];
        }else if(inbits[i] == 1 && inbits[i+1] == 0){
            delta[i] = 1-dprev[1];
            delta[i+1]=dprev[0];
        }else if(inbits[i] == 1 && inbits[i+1] == 1){
            delta[i] = 1-dprev[0];
            delta[i+1]=1-dprev[1];
        }
        dprev[0] = delta[i];
        dprev[1] = delta[i+1];
    }
}

void diffdec(int *bits, int *delta, int nbits){
    int dprev[2]={0,0};
    for(int i=0; i<nbits-1; i+=2){
        if(dprev[0] == 0 && dprev[1] == 0){
            bits[i]   = delta[i];
            bits[i+1] = delta[i+1];
        }else if(dprev[0] == 0 && dprev[1] == 1){
            bits[i]   = 1-delta[i+1];
            bits[i+1] = delta[i];
        }else if(dprev[0] == 1 && dprev[1] == 0){
            bits[i]   = delta[i+1];
            bits[i+1] = 1-delta[i];
        }else if(dprev[0] == 1 && dprev[1] == 1){
            bits[i]   = 1-delta[i];
            bits[i+1] = 1-delta[i+1];
        }
        dprev[0] = delta[i];
        dprev[1] = delta[i+1];
    }
}

void bits2sym(complx *syms, int *bits, int nsym, int bps, complx *lut){
    int ind;
    for(int i=0; i<nsym; i++){
        ind = bits[i*2]*2 + bits[i*2+1];
        syms[i] = lut[ind];
    }
}
```

```
void srrcDelay(float *p, float alpha, float N, int Lp, float Ts, float tau, int rev){
    int i, len = 2*Lp*N+1;
    float *n;
    n =  (float*)calloc(len, sizeof(float));
    for(i=0; i<len; i++){
        n[i] = i - Lp*N - tau;
    }
    for(i=0; i<len; i++){
        if(n[i]*Ts/N == 0){
            p[i] = (1+alpha*(4/PI - 1));
        }else if(n[i]*Ts/N == Ts/(4*alpha) || n[i]*Ts/N == -Ts/(4*alpha)){
            p[i] = alpha*((1+2/PI)*sin(PI/(4*alpha))+(1-2/PI)*(cos(PI/(4*alpha))))/sqrt(2);
        }
        else{
            p[i] = (sin(PI*(1-alpha)*n[i]/N) + (4*alpha*n[i]/N)*cos(PI*(1+alpha)*n[i]/N))
                                                / ((n[i]*PI/N)*(1-pow((4*alpha*n[i]/N),2)));
        }
        p[i] = p[i]/sqrt(N);
    }
    if(rev == 1){
        memcpy(n, p, sizeof(float)*len);
        for(int i=0; i<len; i++){
            p[i] = n[len-1-i];
        }
    }
    free(n);
}

void conv(float *retArr, float *filter, float *dataArr, int filtLen, int dataLen){
    int convLen = dataLen+filtLen-1;
    float *buff;
    buff = (float*)calloc(convLen, sizeof(float));
    memcpy(buff, dataArr, sizeof(float)*dataLen);
    double mac;
    for(int n=0; n<convLen; n++){
        mac = 0.0;
        for(int m=0; m<filtLen; m++){
            mac+=(double)filter[m]*(double)buff[(n-m+convLen)%convLen];
        }
        retArr[n] = (double)mac;
    }
    free(buff);
}

void decisionBlk(int *bits, float *isyms, float *qsyms, complx *lut, int nsym,  int bps){
    float dst, min;
    int bin[2];
    int ind;
    int k = 0;
    for(int i=0; i<nsym; i++){
        min = 1e3; ind = -1;
        for(int j=0; j<4; j++){
            dst = dstCmplx(isyms[i], qsyms[i], lut[j].re, lut[j].im);
            assert(dst>=0);
            if(dst < min){
```

```
                min = dst;
                ind = j;
            }
        }
        switch(ind){
            case 0: bits[k] = 0; bits[k+1] = 0; break;
            case 1: bits[k] = 0; bits[k+1] = 1; break;
            case 2: bits[k] = 1; bits[k+1] = 0; break;
            case 3: bits[k] = 1; bits[k+1] = 1; break;
            default: bits[k] = -1; bits[k+1] = -1; break;
        }
        k+=2;
    }
}

void boxmuller(float *noise, int len, float mean, float var){
    float u1, u2, z1, z2;
    float nstd = sqrt(var);
    assert(len%2 == 0);
    for(int i=0; i<len/2; i++){
      u1 = rand_float();
      u2 = rand_float();
      z1 = sqrt(-2*log(u1))*cos(2*PI*u2);
      z2 = sqrt(-2*log(u1))*sin(2*PI*u2);
      noise[i*2] = (z1 + mean)*nstd;
      noise[i*2+1]=(z2 + mean)*nstd;
    }
}

float dstCmplx(float srcReal, float srcImag, float dstReal, float dstImag){
    float distance;
    distance = sqrt((srcReal-dstReal)*(srcReal-dstReal) + (srcImag-dstImag)*(srcImag-dstImag));
    return distance;
}

float absCmplx2(float numReal, float numImag){  //squared magnitude of complex #
    return numReal*numReal + numImag*numImag;
}

float absCmplx(float numReal, float numImag){   //magnitude of complex #
    return sqrt(numReal*numReal + numImag*numImag);
}

float kurtCmplx(float *datReal, float *datImag, int len){
    float ym2_sum, ym4_sum, ym2;
    float eym4, eym2, ey2m, k;
    float scale = 1/(float)len;
    complx y2, y2_sum, ey2m2;
    ym2 = 0;                                        //|y|^2
    ym2_sum = 0;                                    //sum(|y|^2)
    ym4_sum = 0;                                    //sum(|y|^4)
    y2.re = 0; y2.im = 0;                           //y^2
    y2_sum.re = 0; y2_sum.im = 0;                   //sum(y^2)

    for(int i=0; i<len; i++){
```

```
        ym2 = absCmplx2(datReal[i], datImag[i]);        //|y|^2
        ym2_sum += ym2;                                 //accumulate |y|^2
        ym4_sum += (ym2*ym2);                           //accumulate |y|^4
        y2 = cmpSq(datReal[i], datImag[i]);             //y^2
        y2_sum = cmpAdd(y2_sum, y2);                    //accumulate y^2
    }
    eym4 = scale*ym4_sum;                               //E{|y|^4}
    eym2 = scale*ym2_sum;                               //E{|y|^2}
    ey2m2.re = scale*y2_sum.re;                         //E{y^2}
    ey2m2.im = scale*y2_sum.im;
    ey2m = absCmplx2(ey2m2.re, ey2m2.im);               //|E{y^2}|^2
    k = eym4 - 2*(eym2*eym2) - ey2m;                    //E{|y|^4} - 2(E{|y|^2})^2 - |E{y^2}|^2
    return k;
}

float kurtReal(float *data, int len){
    float scale = 1/(float)len;
    float y2, y2_sum, y4_sum;
    float k;
    y2_sum = 0; y4_sum = 0;
    for(int i=0; i<len; i++){
        y2 = data[i]*data[i];
        y2_sum += y2;
        y4_sum += y2*y2;
    }
    y2_sum = y2_sum * scale;
    k = scale*y4_sum - 3*(y2_sum*y2_sum);
    return k;
}

float deg2rad(float angle){
    return angle*PI/180.0;
}

float rad2deg(float phase){
    return phase*180.0/PI;
}

float getPower(float *data, int len){
   float pow = 0;
   for(int i=0; i<len; i++){
       pow += data[i]*data[i];
   }
   return sqrt(pow/len);
}

complx cmpSq(float numReal, float numImag){      //square of complex #
    complx retVal;
    retVal.re = numReal*numReal - numImag*numImag;
    retVal.im = 2*numReal*numImag;
    return retVal;
}


complx cmpAdd(complx num1, complx num2){         //addition of complex #s
```

```
    complx retVal;
    retVal.re = num1.re+num2.re;
    retVal.im = num1.im+num2.im;
    return retVal;
}

float rand_float(){
  return (float)rand()/RAND_MAX;
}
```

A.2 GPU Functions

gpu.h

```
#ifndef GPU_H
#define GPU_H

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#include "cufft.h"

typedef struct{
    int T;          //sample period
    int N;          //upsampling factor
    int nsym;       //# of symbols (needed?)
    int bps;        //bits per symbol
    float fc;       //carrier frequency
    float alpha;    //excess bandwidth
    int  Lp;        //pulse truncation length
    int sigLen;     //length of signal
    float pow;      //signal power
}signalHdr;

__global__ void dev_convMult(float *convArr, float *filter, float *dataArr, int filtLen, int dataLen, int sigLen);
//Performs multi-threaded convolution
//filter:  pointer to the filter array
//dataArr: pointer to the data array
//filtLen: length of the filter
//dataLen: length of the data array
//sigLen:  length of the return array (= to filtLen + dataLen - 1)

__global__ void dev_conv(float *convArr, float *filter, float *dataArr, int filtLen, int dataLen, int sigLen);
//Performs "inner product" convolution
//convArr: pointer to the return array.
//holds the results of the convolution
//filter:  pointer to the filter array
//dataArr: pointer to the data array
//filtLen: length of the filter
//dataLen: length of the data array
//sigLen:  length of the return array (= to filtLen + dataLen - 1)
```

```
__global__ void dev_srrcDelay(float *pulse, float alpha, float N, int Lp, float Ts, float tau, int rev);
//Generates a square root raised cosine pulse with a timing delay
//pulse : pointer to the return array
//alpha : excess bandwidth parameter
//N     : upsampling factor
//Lp    : truncation length
//Ts    : sample period
//tau   : time delay parameter
//rev   : "reversal" flag: if 1, the pulse array is reversed

__global__ void dev_cmplxPow4(float *data, float *yr, float *yi, int len);
//Calculates the complex 4th power of the re/im input ararys
//data  : pointer to the return array. The real/imaginary outputs are interleaved.
//                      Should be allocated in main with 2*len*sizeof(float) bytes
//yr    : pointer to input array containing the 'real' portion of the data
//yi    : pointer to input array containing the 'imaginary' portion of the data
//len   : number of complex pairs

__global__ void dev_magComplx(float *mag, cufftComplex *data, int len);
//Computes the complex magnitude of the input data array
//mag   : pointer to the output magnitude array
//data  : pointer to the input array of complex data
//len   : number of complex points in the input array

__global__ void dev_initArr(float *data, int len);
//Initializes the elements of data to 0
//data  : pointer to array which is to be initialized to 0
//len   : number of points in the array

__global__ void dev_initArr(cufftComplex *data, int len);
//Initializes the complex elements of data to 0
//data  : pointer to array which is to be initialized to 0
//len   : number of complex points in the array

__global__ void dev_demix(float *sbb, float *spb, int sigLen, int inPhase, float arg, float ph_off);
//Performs the de-mixing step on the received signal
//sbb   : pointer to the return array (signal at baseband)
//spb   : pointer to the input array (signal at passband)
//sigLen: number of points in the signal
//inPhase : flag for in-phase/quad-phase. 1: in-phase portion; 0: quad-phase portion
//arg   : argument of the sin/cos (calculate in main to avoid computation in kernel)
//ph_off: phase offset

__global__ void dev_downsample(float *syms, float *upsamp, int len, int offs, int N);
//Performs the downsampling step on the filtered signal
//syms  : pointer to the return array of symbols
//upsamp: pointer to input array of upsampled data
//len   : number of output symbols
//offs  : number of transient points at the beginning (to be tossed)

__global__ void dev_cfo(float *ups_it, float *ups_qt, float *I, float *Q, float arg, int len);
//De-mixes the carrier frequency offset (rotation by a frequency term)
//ups_it: pointer to returned upsampled in-phase data array
//ups_qt: pointer to returned, upsampled quad-phase data array
```

```
//I      : pointer to the in-phase portion of the filtered signal
//Q      : pointer to the quad-phase portion of the filtered signal
//arg    : argument of the sin/cos (calculate in main to avoid computation in kernel)
//len    : length of the input signal

__global__ void dev_rotate(float *xr, float *yr, float *I, float *Q, float phi, int len);
//Performs the rotation of the symbols by an angle
//xr     : pointer to the returned, rotated in-phase data array
//yr     : pointer to the returned, rotated quad-phase data array
//I      : pointer to the in-phase portion of the input symbols
//Q      : pointer to the quad-phase portion of the input symbols
//phi    : angle by which the array is rotated
//len    : length of the input signal

__global__ void dev_getMin(float *minArr, int *indArr, float *array, int len);
//Finds the local minimum of an array. Reduction is performed using shared memory,
//       so it should be run twice (the second time, blocks per grid should be = 1)
//minArr: pointer to the returned array of local minimums
//indArr: pointer to the returned array of the indexes of the local minimums
//array : pointer to the input array to be searched
//len   : length of the input array
//-----------------------------------------Usage Example-----------------------------------------//
//     dev_getMin<<<blocksPerGrid, threadsPerBlk>>>(dev_minArr, dev_idxArr, dev_dataArr, nfilt);
//     dev_getMin<<<1,             threadsPerBlk>>>(dev_min, dev_idx, dev_minArr, blocksPerGrid);
//       then, dev_idx holds the [location in indArr] of the [location of the minimum value in the input]
//       So: [minimum,ind] = min(dev_dataArr) -->     ind = dev_idxArr[dev_idx]
//                                                minimum = dev_dataArr[ind]

__global__ void dev_getMax(float *maxArr, int *indArr, float *array, int len);
//Finds the local maximum of an input array. Works in the same way as the getMin function.
//maxArr: pointer to the returned array of local maximums
//indArr: pointer to the returned array of the indexes of the local maximums
//array : pointer to the input array to be searched
//len   : length of the input array

__global__ void dev_getSum(float *result, float *data,  int nsym);
//Finds the local (partial) sums of an input array
//result: pointer to the returned array of local sums
//data  : pointer to the input data array to be summed
//nsym  : number of points in the input array
//Usual usage: if blocks per grid is 1, then *result is a single number.

__global__ void dev_cmplxKSums(float *ym4Arr, float *ym2Arr, float *y2rArr, float *y2iArr, float *real,
                                                                                    float *imag, int nsym);
//Computes the 'intermediate' sums for the values used in the complex kurtosis.
//                              Operates on a complex input array; so y = a + jb
//ym4Arr: pointer to the output array for |y|^4
//ym2Arr: pointer to the output array for |y|^2
//y2rArr: pointer to the real portion of the output array for y^2
//y2iArr: pointer to the imag portion of the output array for y^2
//real  : pointer to the real portion of the input
//imag  : pointer to the imag portion of the input
//nsym  : number of complex symbols (aka length of the input arrays)
//Usage: calculate, then send each 'intermediate sum' array through the getSum function
```

```
__global__ void dev_cmplxKurt(float *kurt, float *ym4, float *ym2, float *y2r, float *y2i, int nsym);
//Computes the complex kurtosis given the sum of the values used in the kurtosis
//kurt  : pointer to the output (the calculated kurtosis) which is equal to a single float value
//ym4   : pointer to the sum of |y|^4
//ym2   : pointer to the sum of |y|^2
//y2r   : pointer to the real portion of the sum of y^2
//y2i   : pointer to the imag portion of the sum of y^2
//nsym  : number of symbols in the original array for which the kurtosis is calculated
//      (used for finding expected values)

__global__ void dev_mult(float *res, float *m1, float *m2, int len);
//Multiplies the elements of two arrays and returns their bit-wise product
//res   : pointer to the return array (results)
//m1    : pointer to the first array to be multiplied
//m2    : pointer to the second array to be multiplied
//len   : length of the input arrays

__global__ void dev_multCmplx(cuComplex *result, cuComplex *m1, cuComplex *m2, int len);
//Multiplies two arrays of complex numbers (a+jb)*(c+jd) = ac-bd + j*(ad + bc)
//result: pointer to the complex result of the multiplication
//m1    : pointer to the first array to be multiplied
//m2    : pointer to the second array to be multiplied; assumed to be a half-length
//                          array of complex numbers (e.g. the fft of a real signal)
//len   : length of the first input array (second input is of length len/2)

__global__ void dev_multConst(float *res, float *in, float scale, int len);
//Performs a scaling operation on the input array (multiplication by a constant value)
//res   : pointer to the returned result of the scaling
//in    : pointer to the input array to be scaled
//scale : value by which the array is multiplied
//len   : length of the input array

__global__ void dev_multConst(cuComplex *res, cuComplex *in, float scale, int len);
//Performs a scaling operation on the complex input array (multiplication by a constant value)
//res   : pointer to the returned complex result of the scaling
//in    : pointer to the complex input array to be scaled
//scale : value by which the array is multiplied
//len   : length of the input array

__global__ void dev_abs(float *data, int len);
//Returns the absolute value of the input data

#endif
```

gpu.cu

```
#include "gpu.h"

#define PI 3.14159265358979323846

__global__ void dev_fastConv(cuComplex *result, cuComplex *m1, cuComplex *m2, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int mid = tid;
    int step = blockDim.x*gridDim.x;
    float scale = 1/(float)len;
    while(tid < len){
        if(tid > len/2)mid = tid - (len/2);
        result[tid].x = scale*(m1[tid].x*m2[mid].x - m1[tid].y*m2[mid].y);
        result[tid].y = scale*(m1[tid].y*m2[mid].x + m1[tid].x*m2[mid].y);
        tid += step;
    }
}

__global__ void dev_convMult(float *convArr, float *filter, float *dataArr, int filtLen, int dataLen, int sigLen){
    int tid = blockIdx.x*blockDim.x; //thread index
    int l  = threadIdx.x;            //accumulator index
    int i  = 0;                      //current time index
    int m  = 0;                      //coefficient index
    int N  = filtLen;                //size of block and # of accumulators
    int str = blockDim.x*blockIdx.x;
    int end = blockDim.x*(blockIdx.x+1)+N;
    __shared__ float filt[1024];
    if(l<N)filt[l] = filter[l];
    float x;
    register float acc = 0.f;
    while(tid < end){
        m = (l-i+N)%N;
        if(tid < dataLen && tid >=str)
            x = dataArr[tid];
        else
            x = 0.0;
        acc += filt[m]*x;
        if(m == 0){
            if(tid>= str && tid < sigLen){
                convArr[tid] = acc;
                acc = 0.0;
            }
        }
        i = (i+1+N)%N;
        tid++;
    }
}

__global__ void dev_conv(float *convArr, float *filter, float *dataArr, int filtLen, int dataLen, int sigLen){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int convLen = filtLen + dataLen - 1;
    int step = blockDim.x*gridDim.x;
    int l = threadIdx.x;
    __shared__ float filt[1024];              //put the filter in shared memory; max threads/block is 1024
```

```
    float tmp, x;
    int ind;
    while(l < filtLen){
        filt[l] = filter[l];
        l += blockDim.x;
    }
    __syncthreads();
    while(tid < convLen){
        tmp = 0.0;
        for(int i=0; i<filtLen; i++){
            ind = (tid - i + sigLen)%sigLen; //% is expensive
            if(ind > dataLen - 1) x = 0;
            else x = dataArr[ind];
            tmp += filt[i]*x;
        }
        convArr[tid] = tmp;
        tid += step;
    }
}

__global__ void dev_srrcDelay(float *pulse, float alpha, float N, int Lp, float Ts, float tau, int rev) {
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int len = 2*Lp*N+1;
    int step = blockDim.x*gridDim.x;
    float n, p;
    while(tid < len){
        n = tid - Lp*N - tau;
        if(n*Ts/N == 0) {
            p = (1+alpha*(4/PI - 1));
        }else if(n*Ts/N == Ts/(4*alpha) || n*Ts/N == -Ts/(4*alpha)){
            p = alpha*((1+2/PI)*sinf(PI/(4*alpha))+(1-2/PI)*(cosf(PI/(4*alpha))))/sqrtf(2);
        }else{
            p = (sinf(PI*(1-alpha)*n/N) + (4*alpha*n/N)*cosf(PI*(1+alpha)*n/N))
                                        / ((n*PI/N)*(1-powf((4*alpha*n/N),2)));
        }
        p = p/sqrtf(N);
        if(rev == 1){
            pulse[len-1-tid] = p;
        }else{
            pulse[tid] = p;
        }

        tid += step;
    }
}

__global__ void dev_cmplxPow4(float *data, float *yr, float *yi, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    float y2r, y2i, y4r, y4i;
    int step = blockDim.x*gridDim.x;
    while(tid < len){                               //len is the # of complex pairs
        y2r = yr[tid]*yr[tid] - yi[tid]*yi[tid];    //Re(y^2): a^2 - b^2
        y2i = 2*yr[tid]*yi[tid];                    //Im(y^2): 2*a*b
        y4r = y2r*y2r - y2i*y2i;                    //Re(y^4): re(y^2)^2 - im(y^2)^2
        y4i = 2*y2r*y2i;                            //Im(y^4): 2*re(y^2)*im(y^2)
```

```
        data[tid*2]   = y4r;                        //real/imaginary portions interleaved
        data[tid*2+1] = y4i;

        tid += step;
    }
}

__global__ void dev_magComplx(float *mag, cufftComplex *data, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    while(tid < len){
        mag[tid] = data[tid].x*data[tid].x + data[tid].y*data[tid].y;
        tid += step;
    }
}

__global__ void dev_initArr(float *data, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    while(tid < len){
        data[tid] = 0.0;
        tid += step;
    }
}
__global__ void dev_initArr(cufftComplex *data, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    while(tid < len){
        data[tid].x = 0.0;
        data[tid].y = 0.0;
        tid += step;
    }
}

__global__ void dev_demix(float *sigBaseBand, float *sigPassBand, int sigLen, int inPhase,
                                                                    float arg, float ph_off){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    float sqrt2 = sqrtf(2);
    if(inPhase == 1){       //in-phase portion
        while(tid < sigLen){
            sigBaseBand[tid] = sqrt2*cosf(arg*tid + ph_off)*sigPassBand[tid];
           tid += step;
        }
    }else{                  //quadrature phase portion
        while(tid < sigLen){
            sigBaseBand[tid] =-sqrt2*sinf(arg*tid + ph_off)*sigPassBand[tid];
           tid += step;
       }
    }
}

__global__ void dev_downsample(float *syms, float *upsamp, int len, int offs, int N){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
```

```
    while(tid < len){
        syms[tid] = upsamp[(offs+tid)*N];
        tid += step;
    }
}

__global__ void dev_cfo(float *ups_it, float *ups_qt, float *I, float *Q, float arg, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    float cos_theta, sin_theta;
    int step = blockDim.x*gridDim.x;
    while(tid<len){
        cos_theta = cosf(arg*tid);
        sin_theta = sinf(arg*tid);
        ups_it[tid] = I[tid]*cos_theta - Q[tid]*sin_theta;
        ups_qt[tid] = I[tid]*sin_theta + Q[tid]*cos_theta;
        tid += step;
    }
}

__global__ void dev_rotate(float *xr, float *yr, float *I, float *Q, float phi, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    float C, S;
    int step = blockDim.x*gridDim.x;
    C = cosf(phi); S = sinf(phi);
    while(tid<len){
        xr[tid] = C*I[tid] - S*Q[tid];
        yr[tid] = S*I[tid] + C*Q[tid];
        tid += step;
    }
}

__global__ void dev_getMin(float *minArr, int *indArr, float *array, int len){
    __shared__ float minCache[1024];
    __shared__ float indCache[1024];
    int cacheIdx = threadIdx.x;
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    float min = 1e10;
    int ind = -1;
    while(tid<len){
        if(array[tid] < min){
            min = array[tid];
            ind = tid;
        }
        tid += step;
    }

    minCache[cacheIdx] = min;
    indCache[cacheIdx] = ind;
    __syncthreads();

    int i = blockDim.x/2;
    //Do the reduction
    while(i!=0){
        if(cacheIdx < i){
```

```
            if(minCache[cacheIdx+i] < minCache[cacheIdx]){
                //printf("%d\n", cacheIdx);
                minCache[cacheIdx] = minCache[cacheIdx+i];
                indCache[cacheIdx] = indCache[cacheIdx+i];
            }
        }
        __syncthreads();
        i/=2;
    }
    __syncthreads();
    if(cacheIdx == 0){
        minArr[blockIdx.x] = minCache[0];
        indArr[blockIdx.x] = indCache[0];
    }
}

__global__ void dev_getMax(float *maxArr, int *indArr, float *array, int len){
    __shared__ float maxCache[1024];
    __shared__ float indCache[1024];
    int cacheIdx = threadIdx.x;
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    float max = -1e10;
    int ind = -1;
    int step = blockDim.x*gridDim.x;
    while(tid<len){
        if(array[tid] > max){
            max = array[tid];
            ind = tid;
        }
        tid += step;
    }

    maxCache[cacheIdx] = max;
    indCache[cacheIdx] = ind;
    __syncthreads();

    int i = blockDim.x/2;
    //Do the reduction
    while(i!=0){
        if(cacheIdx < i){
            if(maxCache[cacheIdx+i] > maxCache[cacheIdx]){
                maxCache[cacheIdx] = maxCache[cacheIdx+i];
                indCache[cacheIdx] = indCache[cacheIdx+i];
            }
        }
        __syncthreads();
        i/=2;
    }
    __syncthreads();
    if(cacheIdx == 0){
        maxArr[blockIdx.x] = maxCache[0];
        indArr[blockIdx.x] = indCache[0];
    }
}
```

```
__global__ void dev_getSum(float *result, float *data,  int nsym){
    // Sum the elements of data; put partial results in result
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int cacheIdx = threadIdx.x;
    float localSum = 0;
    int step = blockDim.x*gridDim.x;
    __shared__ float cache[1024];
    while(tid < nsym){
        localSum += data[tid];
        tid += step;
    }

    cache[cacheIdx] = localSum;      //set cache values
    __syncthreads();                 //sync threads before accessing cache data

    int i = blockDim.x/2;            //assume threads per block is a power of 2
    while(i!=0){
        if(cacheIdx < i){
            cache[cacheIdx] += cache[cacheIdx+i];
        }
        __syncthreads();
        i/=2;
    }
    __syncthreads();
    if(cacheIdx == 0){
        result[blockIdx.x] = cache[0];
    }
}

__global__ void dev_cmplxKSums(float *ym4Arr, float *ym2Arr, float *y2rArr, float *y2iArr,
                                                              float *real, float *imag, int nsym){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int cacheIdx = threadIdx.x;
    int step = blockDim.x*gridDim.x;
    __shared__ float ym4Cache[1024];
    __shared__ float ym2Cache[1024];
    __shared__ float y2rCache[1024];
    __shared__ float y2iCache[1024];
    float ym4, ym2, y2r, y2i, mag;
    ym4 = 0.0; ym2 = 0.0; y2r = 0.0; y2i = 0.0;
    while(tid < nsym){
        mag  = real[tid]*real[tid] + imag[tid]*imag[tid];
        ym4 += mag*mag;
        ym2 += mag;
        y2r += real[tid]*real[tid] - imag[tid]*imag[tid];
        y2i += 2.0*real[tid]*imag[tid];
        tid += step;
    }
    ym4Cache[cacheIdx] = ym4;
    ym2Cache[cacheIdx] = ym2;
    y2rCache[cacheIdx] = y2r;
    y2iCache[cacheIdx] = y2i;
    __syncthreads();

    int i = blockDim.x/2;
```

```
    while(i!=0){
        if(cacheIdx < i){
            ym4Cache[cacheIdx] += ym4Cache[cacheIdx+i];
            ym2Cache[cacheIdx] += ym2Cache[cacheIdx+i];
            y2rCache[cacheIdx] += y2rCache[cacheIdx+i];
            y2iCache[cacheIdx] += y2iCache[cacheIdx+i];
        }
        __syncthreads();
        i/=2;
    }
    __syncthreads();
    if(cacheIdx == 0){
        ym4Arr[blockIdx.x] = ym4Cache[0];
        ym2Arr[blockIdx.x] = ym2Cache[0];
        y2rArr[blockIdx.x] = y2rCache[0];
        y2iArr[blockIdx.x] = y2iCache[0];
    }
}

__global__ void dev_cmplxKurt(float *kurt, float *ym4, float *ym2, float *y2r, float *y2i, int nsym){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    float k;
    if(tid == 0){
        float eym4, eym2, ey2m;
        eym4 = ym4[0]/(float)nsym;
        eym2 = (ym2[0]/(float)nsym)*(ym2[0]/(float)nsym);
        ey2m = (y2r[0]/(float)nsym)*(y2r[0]/(float)nsym) + (y2i[0]/(float)nsym)*(y2i[0]/(float)nsym);
        k = eym4 - 2*eym2 - ey2m;
        kurt[0] = k;
    }
}

__global__ void dev_mult(float *res, float *m1, float *m2, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    while(tid < len){
        res[tid] = m1[tid]*m2[tid];
        tid += step;
    }
}

__global__ void dev_multCmplx(cuComplex *result, cuComplex *m1, cuComplex *m2, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int mid = tid;
    int step = blockDim.x*gridDim.x;
    while(tid < len){
        if(tid > len/2)mid = tid - (len/2);
        result[tid].x = m1[tid].x*m2[mid].x - m1[tid].y*m2[mid].y;
        result[tid].y = m1[tid].y*m2[mid].x + m1[tid].x*m2[mid].y;
        tid += step;
    }
}

__global__ void dev_multConst(float *res, float *in, float scale, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
```

```
    int step = blockDim.x*gridDim.x;
    while(tid < len){
        res[tid] = in[tid]*scale;
        tid += step;
    }
}

__global__ void dev_multConst(cuComplex *res, cuComplex *in, float scale, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    while(tid < len){
        res[tid].x = in[tid].x*scale;
        res[tid].y = in[tid].y*scale;
        tid += step;
    }
}

__global__ void dev_abs(float *data, int len){
    int tid = blockIdx.x*blockDim.x + threadIdx.x;
    int step = blockDim.x*gridDim.x;
    while(tid < len){
        if(data[tid] < 0)
            data[tid] = -data[tid];
        tid += step;
    }
}
```

APPENDIX B

System Diagrams

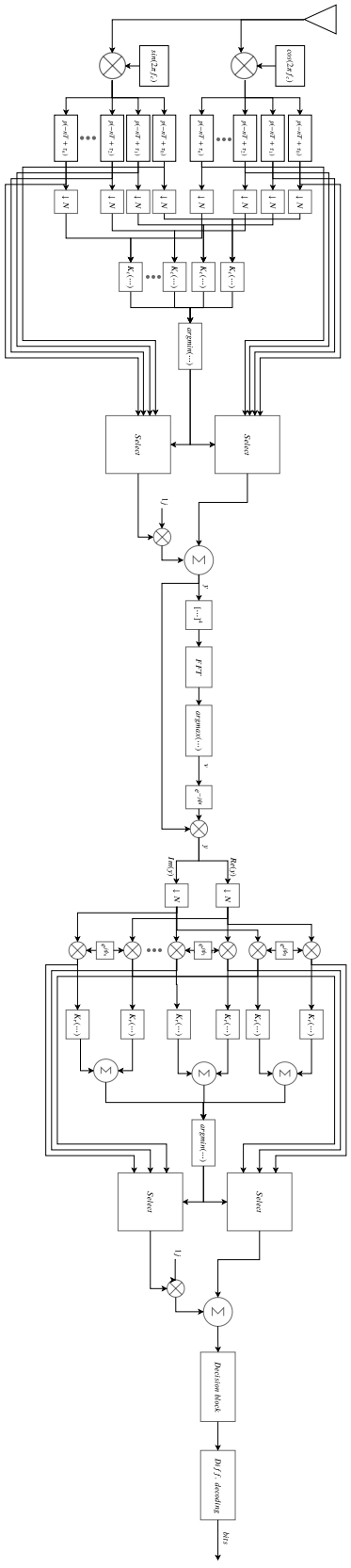


Fig. B.1: Full System Diagram

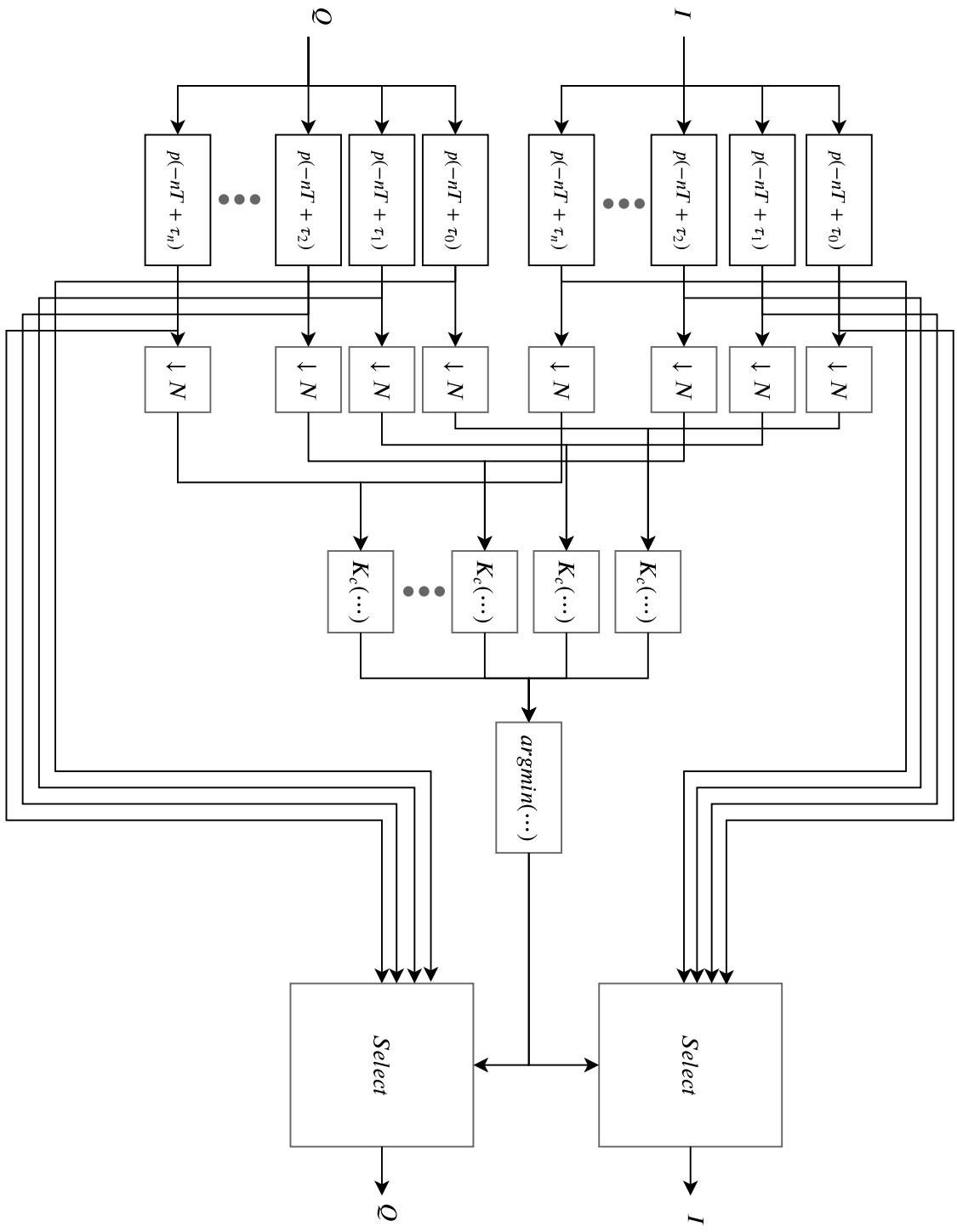


Fig. B.2: Timing Correction System Diagram

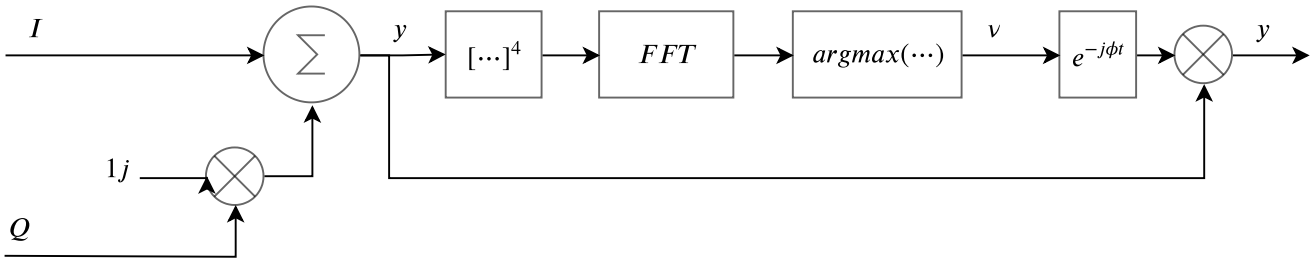


Fig. B.3: Carrier Frequency Offset Correction Diagram

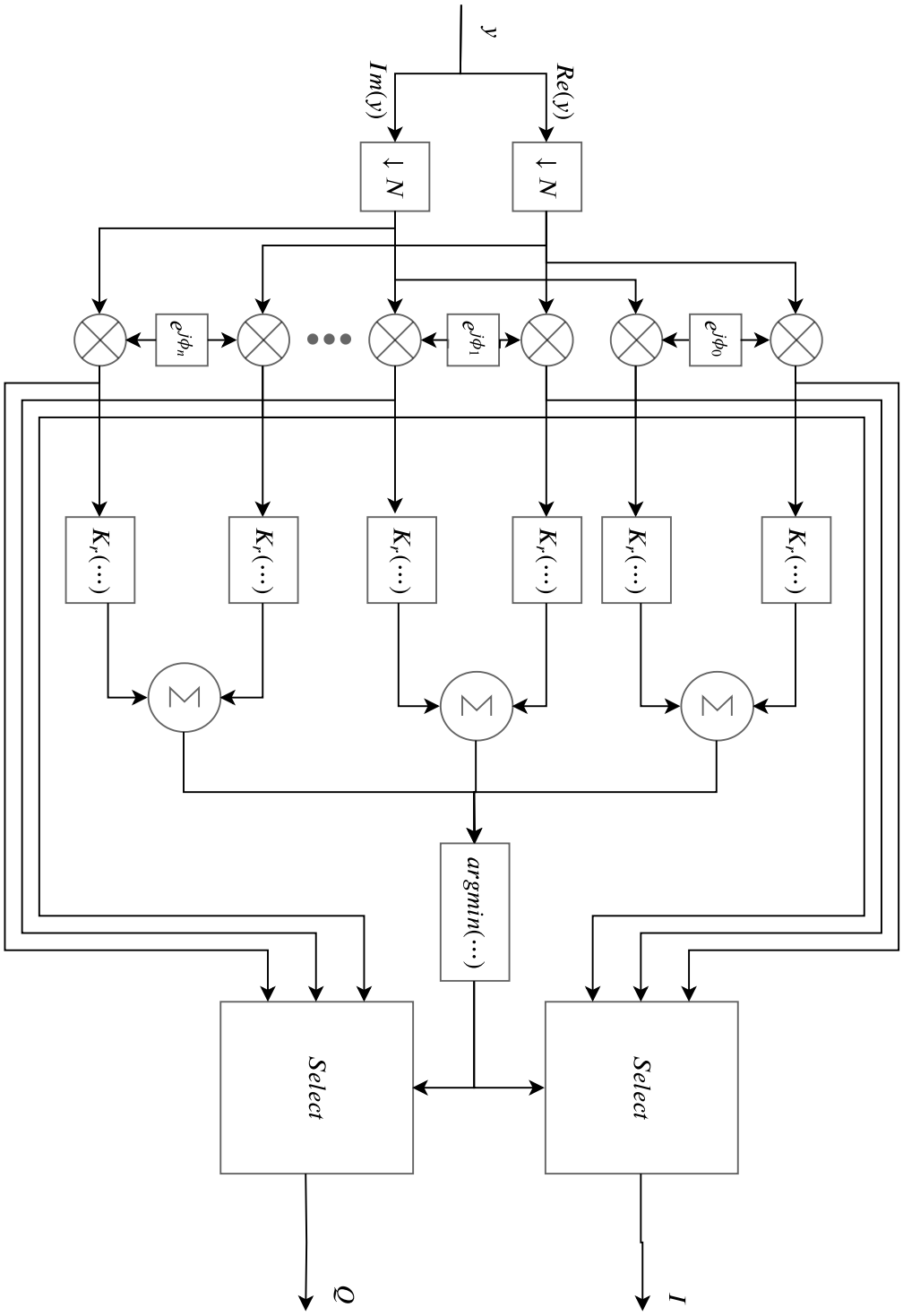


Fig. B.4: Phase Correction System Diagram