ABSTRACT

Title of dissertation:	BASEBAND RADIO MODEM DESIGN USING GRAPHICS PROCESSING UNITS
	Scott C. Kim Doctor of Philosophy, 2015
Dissertation directed by:	Professor Shuvra S. Bhattacharyya Dept. of Electrical & Computer Engineering, and Institute for Advanced Computer Studies

A modern radio or wireless communications transceiver is programmed via software and firmware to change its functionalities at the baseband. However, the actual implementation of the radio circuits relies on dedicated hardware, and the design and implementation of such devices are time consuming and challenging. Due to the need for real-time operation, dedicated hardware is preferred in order to meet stringent requirements on throughput and latency. With increasing need for higher throughput and shorter latency, while supporting increasing bandwidth across a fragmented spectrum, dedicated subsystems are developed in order to service individual frequency bands and specifications. Such a dedicated-hardware-intensive approach leads to high resource costs, including costs due to multiple instantiations of mixers, filters, and samplers. Such increases in hardware requirements in turn increases device size, power consumption, weight, and financial cost.

If it can meet the required real-time constraints, a more flexible and reconfigurable design approach, such as a software-based solution, is often more desir-

able over a dedicated hardware solution. However, significant challenges must be overcome in order to meet constraints on throughput and latency while servicing different frequency bands and bandwidths. Graphics processing unit (GPU) technology provides a promising class of platforms for addressing these challenges. GPUs, which were originally designed for rendering images and video sequences, have been adapted as general purpose high-throughput computation engines for a wide variety of application areas beyond their original target domains. Linear algebra and signal processing acceleration are examples of such application areas.

In this thesis, we apply GPUs as software-based, baseband radios and demonstrate novel, software-based implementations of key subsystems in modern wireless transceivers. In our work, we develop novel implementation techniques that allow communication system designers to use GPUs as accelerators for baseband processing functions, including real-time filtering and signal transformations. More specifically, we apply GPUs to accelerate several computationally-intensive, front-end radio subsystems, including filtering, signal mixing, sample rate conversion, and synchronization. These are critical subsystems that must operate in real-time to reliably receive waveforms.

The contributions of this thesis can be broadly organized into 3 major areas: (1) channelization, (2) arbitrary resampling, and (3) synchronization.

1. Channelization: a wideband signal is shared between different users and channels, and a channelizer is used to separate the components of the shared signal in the different channels. A channelizer is often used as a pre-processing step in selecting a specific channel-of-interest. A typical channelization process involves sig-

nal conversion, resampling, and filtering to reject adjacent channels. We investigate GPU acceleration for a particularly efficient form of channelizer called a polyphase filterbank channelizer, and demonstrate a real-time implementation of our novel channelizer design.

2. Arbitrary resampling: following a channelization process, a signal is often resampled to at least twice the data rate in order to further condition the signal. Since different communication standards require different resampling ratios, it is desirable for a resampling subsystem to support a variety of different ratios. We investigate optimized, GPU-based methods for resampling using polyphase filter structures that are mapped efficiently into GPU hardware. We investigate these GPU implementation techniques in the context of interpolation (integer-factor increases in sampling rate), decimation (integer-factor decreases in sampling rate), and rational resampling. Finally, we demonstrate an efficient implementation of arbitrary resampling using GPUs. This implementation exploits specialized hardware units within the GPU to enable efficient and accurate resampling processes involving arbitrary changes in sample rate.

3. Synchronization: incoming signals in a wireless communications transceiver must be synchronized in order to recover the transmitted data properly from complex channel effects such as thermal noise, fading, and multipath propagation. We investigate timing recovery in GPUs to accelerate the most computationally intensive part of the synchronization process, and correctly align the incoming data symbols in the receiver. Furthermore, we implement fully-parallel timing error detection to accelerate maximum likelihood estimation.

BASEBAND RADIO MODEM DESIGN USING GRAPHICS PROCESSING UNITS

by

Scott C. Kim

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Dedication

to my parents and my wife

Acknowledgements

First and foremost I would like to thank my advisor, Professor Shuvra S. Bhattacharyya for his guidance, support, encouragement, and motivation over the course of my graduate career. I especially am grateful to him for giving me the freedom to develop and pursue my own ideas and topics, sponsoring travels to conferences, making me a better writer through his strict review process, providing insights, and paying attention to details.

I would like to thank my committee members for reviewing my thesis, providing feedback, and accommodating throughout the process. In addition, I would like to thank Professor Joseph Cavallaro of Rice University for the initial collaboration with my research and a visit to Rice University.

I have been fortunate to have so many wonderful teachers and coaches throughout my life and I would like to specially thank Professor fred harris of San Diego State University for introducing me to signal processing and modem design in 2003, and providing motivation to pursue my Ph.D. degree in modem design later on. Without him, I would not be here today. I would also like to thank Jamie Brown, Anthony Casipit, Michael Deveney, Professor Anthony England, Professor Brian Gilchrist, Professor Jeff Fessler, Dr. Brian Jennison, Professor Bruce Jacob, Dr. Martin Johnson, Professor Steve Marcus, David Miller, Professor Gang Qu, Dr. Brian Sadler, Professor Charles Silio, Dr. Bernie Sklar, Leo Totten, Michael Walters, Professor Donald Yeung, and many others whom I may have missed.

I have also been fortunate to work with some very talented colleagues over the years, particularly Kevin Borries, Thomas Chatt, Craig Connacher, Ahmed Fasih, Craig Weaver, Kyle Weber, Michael Wu, and many others.

It has been a pleasure working with members of the DSPCAD group and students from the ECE department at the University of Maryland, especially Inkeun Cho, Wei-Hong Chuang, Ilya Chukhman, Domenic Forte, Jim Gruen, Sang-Kyo Han, Hojin Kee, Shuoxin Lin, Nimish Sane, Chung-Ching Shen, Kishan Sudusinghe, Hsiang- Huang Wu, and George Zaki. I want to specially thank our Research Scientist, Dr. Will Plishker, for introducing me to CUDA in 2010, his help with troubleshooting CUDA issues over the years, and reviewing my early papers that set the foundation for the future papers.

I owe deepest thanks to my family, especially my parents, my brother, and my wife.

Although the Ph.D. process is intense and challenging with so many unknowns, countless hours of researching and frustrations, the past 8 years has been one of the most memorable times of my life. Not only do I feel better prepared for my future career, but as a researcher, I feel I have contributed something to this world. I highly encourage anyone to enter the Ph.D. program and further advance science and engineering!

Lastly, thank you all and thank God for everything.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADC	analog-to-digital conversion
AP	application processor
API	application programming interface
ASIC	application specific integrated circuit
ASRC	arbitrary sample rate conversion
AWGN	additive white Gaussian noise
BB	baseband
BW	bandwidth
CA	carrier aggregation
CM	constant memory
COI	channel-of-interest
CPU	central processing unit
CT	continuous time
CUDA	compute unified device architecture
CUFFT	CUDA fast Fourier transform
DAC	digital-to-analog conversion
DFT	discrete Fourier transform
dMF	derivative matched filter
DT	discrete time
FBMC	filter bank multi-carrier
FDM	frequency division multiplexing
FFT	fast Fourier transform
FPGA	field programmable gate array
FMA	fused multiply-add
FSM	finite state machine
GBE	GPU back-end
GBR	GPU-based radio
GFE	GPU front-end
GM	global memory
GPP	general purpose processor
GPGPU	general purpose graphics processing unit
GPU	graphics processing unit
HDL	hardware description language
IF	intermediate frequency
IP	inner-product
ISI	inter-symbol interference

LF	loop filter
LM	linear memory
LN	linear
LPF	low pass filter
MAC	multiply and accumulate
MC	multi-carrier
MCM	multi-carrier modulation
MF	matched filter
ML	maximum likelihood
NCO	numerically controlled oscillator
NN	nearest neighborhood
OFDM	orthogonal frequency division multiplexing
PDSP	programmable digital signal processor
PFB	polyphase filter bank
PLL	phase locked loop
RF	radio frequency
RFIC	radio frequency integrated circuit
RFFE	radio frequency front-end
RX	receiver
SDR	software-defined radio
SIMD	single instruction multiple data
SIMT	single instruction multiple thread
SM	shared memory
SNR	signal-to-noise ratio
SoC	system-on-chip
SOI	signal-of-interest
SP	signal processing
Sps	samples per second
SRC	sample rate conversion
STR	symbol timing recovery
SWR	software radio
TDM	time division multiplexing
TED	timing error detection
TM	texture memory
TPB	threads per block
TRX	transceiver
TX	transmitter

Chapter 1: Introduction

1.1 Overview

Radios or transceivers for wireless communication have evolved over the decades from being based primarily on analog circuits to their extensive use of digital integrated circuits. Dedicated hardware devices, such as application specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs), have become the dominant platforms for implementing digital radios, which are programmed using firmware and software. At the same time, general purpose processors (GPPs) have become increasingly powerful in terms of computation performance and have became easier to use in terms of programmability.

With such programmability, software has become the preferred method for designing radios and implementing signal processing algorithms in many application contexts, particularly at the baseband (i.e., for signals that have negligible frequency content outside of a relatively narrow range $[0, F_{\max}]$). For such baseband processing, the signal is mixed up or down to or from the desired center frequency by using a radio frequency integrated circuit (RFIC). This isolation of baseband processing from the radio frequency front-end (RFFE) has major implications on the design and implementation of modern radio systems.

For example, a baseband modem engineer can focus on signal processing (SP) algorithm implementation, whereas an RFIC or RFFE designer can focus on the analog and radio frequency (RF) aspects of the design. A common interface, such as a software application programming interface (API), is then defined to facilitate communication between the baseband and RF or "front end" subsystems. This provides the framework for software radio (SWR), where the entire processing chain from baseband to RF is controlled by software. We distinguish this from software-defined radio (SDR), where software only controls the subsystems that encompass baseband and intermediate frequency (IF) processing [1].

However, even with computational performance improvement in terms of speed, number of processing cores, etc., GPPs, such as CPUs, are not geared towards the intense computational demands of many SP applications. For such applications, domain-specific or dedicated accelerators can be employed to improve signal processing performance. For example, programmable digital signal processors (PDSPs) provide single instruction multiple data (SIMD) operations and specialized computation units that are geared towards critical SP functions. SIMD is a form of parallelization where the same instructions or operations are applied simultaneously on different data items.

Graphics processing units (GPUs) belong to another important class of specialized computational devices that are useful for accelerating complex signal processing applications. While GPUs were originally designed to render graphics for images and videos, they are now employed across a broad range of application areas that involve intensive use of regularly structured computations. GPUs are capable of exploiting large amounts of SIMD parallelism, and can be applied for acceleration of single- and multi-dimensional signal processing operations.

With the introduction of general purpose GPU (GPGPU) languages, such as CUDA [2], GPUs have been gaining significant interest in the SP systems community. GPUs are attractive for SP systems, for example, as an alternative to hardware implementation. With this motivation, we explore GPGPUs as platforms for design and implementation of real-time baseband *radio modems*, particularly as front-end transceiver processors to accelerate filtering and transform algorithms commonly found in SP applications.

A radio modem subsystem is a dedicated unit for servicing different radio specifications, such as standards for wireless cellular communication, navigation, and wireless local area networking. Each of these standards is typically implemented to operate across different ranges of communication and signal processing parameters, including frequency bands, bandwidths, sampling rates, data rates, modulation types, coding rates, etc.

In this thesis, we address the challenges of accelerating SP algorithms via software, particularly using GPUs as parallel transceivers to help realize real-time radio modems, while providing the flexibility and cost-efficiency of software solutions. Our goal is to use the GPU as (1) a complete modem or (2) a coprocessor that works in conjunction with an existing hardware modem to extend its functionality at the physical layer. In this thesis, we focus on the receiver architecture, and more specifically, on baseband operations within the receiver. We emphasize methods for efficient implementation of wideband receivers — i.e., receivers that process bandwidths greater than hundreds of MHz, and that include subsystems for channelization, resampling, and synchronization.

1.2 Contributions

In this thesis, we develop a novel transceiver architecture using GPUs, particularly as a parallel front-end transceiver that performs most of the required resampling, signal conversion, and transformation for the enclosing transceiver system. Although, our methods apply to both transmitter and receiver design, we emphasize in our work the receiver design aspect due to the challenges involved in recovering the transmitted signal from channel impairments.

The contributions of this thesis are presented in four main parts. First, we introduce the application of GPUs to front-end transceiver implementation, and we examine the subsystems that comprise a wideband GPU front-end. We are presented with baseband discrete-time signals immediately after the sampling circuit in the RFFE. The transmitted original signal is corrupted and we must recover it from the channel impairments. We perform a channelization or separation of the users or channels from the wide composite signal that is made of multiple subcarriers. Following this channelization, the signal must be resampled properly to multiples of its data rate for further signal processing, such as synchronization. Synchronization is one of the most critical aspects of receiver design in that it must estimate frequency, phase, and time offsets and correct them in order to align the received signal to the transmitted signal. Following these front-end stages, further signal processing, such as channel and source decoding, is performed in the back-end receiver [3, 4].

1.2.1 GPU Front-end Transceiver

Our goal of designing a GPU-based radio can be broken into the design of two major sub-systems: the front-end and back-end. A front-end receiver or "inner-receiver" is responsible for filtering, mixing the signals, and resampling to multiples of the desired data rate for back-end receiver processing [3, 4]. We generalize this notion with the design of a wideband receiver using a channelizer to filter, down-convert, and downsample the channel-of-interest (COI). Then, if the sampling rate out of the channelizer does not meet the desired rate (typically 2–4 times the data rate), we perform further resampling.

A major objective in our work is to employ GPU-based parallel processing to accelerate the front-end radio computations, which are typically implemented in dedicated hardware. We perform all acceleration using highly optimized GPU software, and demonstrate that our resulting designs exceed the throughput and latency requirements for real-time operation. By implementing a complete set of front-end radio functions in a GPU, we demonstrate the concept of a *GPU front-end (GFE)* receiver. This is a type of GPU-based radio (GBR) that consists of a GFE and GPU back-end (GBE). The GBE is largely responsible for channel and source decoding.

As part of our work on GPU-based transceiver design, we adapt efficient channelization methods using a polyphase filter bank (PFB) structure and a GPU-optimized discrete Fourier transform (DFT) library kernel, called CUFFT. We apply our resulting architecture to efficient channelization of 2G GSM signals. We also demonstrate arbitrary resampling of all channels simultaneously to fractionally resample the channelizer output so that it meets the fractional GSM data rate. With our optimized GPU implementation, all of these operations execute within the radio frame duration, and also, the throughput is increased significantly due to our ability to process hundreds of channels simultaneously using a single GPU.

This work on the GFE as an efficient arbitrary resampling channelizer is presented in Chapter 3.

1.2.2 Wideband Channelization

In order to decrease the time required to process wireless communication radio frames, GFE implementation needs to be optimized carefully to reduce latency. One method for reducing latency is to spread the workload across the GPU more effectively, which in turn increases occupancy since more threads are kept busy during GPU kernel calls. In order to increase the occupancy, we assign a single output sample to a single thread of a GPU kernel. This is in contrast to our GFE design mentioned earlier, which parallelizes the filter operation, where a thread is responsible for a single multiplication operation between the input sample and a filter coefficient. The accumulation is performed across multiple threads, which pauses the other threads.

The initial GFE design contains serial loops to index through the input samples. In contrast, in our new channelizer design, we completely unroll the loops, which eliminates serial loops in the kernel. Since there are more samples to process than the filter length, a large number of threads are instantiated, where each thread performs the same instruction across the entire kernel, utilizing the GPU's SIMD operations. The resulting architecture provides significant improvement in throughput, and reduces the latency as well. We apply our methods to 3G WCDMA signals and demonstrate their performance in this practical context.

This work on a new high-throughput, low-latency, fully parallel polyphase channelizer design is presented in Chapter 4.

1.2.3 Multi-channel Arbitrary Resampling

From our motivation to significantly increase throughput and reduce latency, we examine in depth the resampling architecture of our GFE. To optimize the design of this architecture, we apply a distinctive hardware feature in GPUs called a texture unit. A texture unit has its own memory, called texture memory (TM), and has a built-in interpolation circuit to perform fast interpolation directly while fetching the data from TM. A texture unit provides two alternative modes of operation for interpolation — nearest neighborhood (NN) and linear (LN) interpolation.

We utilize the GPU texture unit to develop a novel design for a low-complexity and low-latency arbitrary resampler. In order to increase the arbitrary resampling resolution, we provide an integer interpolation prior to the TM kernel call. We perform two different integer interpolations using time and frequency domain filtering. CUDA provides a highly optimized and parallelized DFT called CUFFT. We adapt CUFFT to perform integer interpolation via zero-padding in the frequency domain and use its sinc waveform to provide filter-free interpolation in the time domain. We then replace the CUFFT interpolation with a more traditional time domain integer interpolation, which allows us to design our optimized filter in the time domain, and without the constraints involving prime factor ratios that are imposed in CUFFT interpolation.

We adapt a polyphase interpolator for arbitrary resampling and expand its filtering capability beyond one-dimensional (1D) signal processing by expanding the subsystem to process multiple channels and full radio frames using a 2D polyphase structure. The idea is further expanded to 3D by adding different frequency bands containing different channels. This multi-dimensional approach to process multiple bands, channels, and full radio frames provides a novel application of the multi-dimensional parallelism of GPUs to an important class of computationally-intensive, wireless communications subsystems. This approach can be viewed as a type of carrier aggregation (CA), which is an attractive option for processing wideband channels in current- and future-generation systems.

We integrate our multi-channel interpolator with a TM to arbitrarily resample the data at any desired rate. This allows the decomposition of input radio frames into successive slots that are processed simultaneously. The parallelization across slots within a frame helps to overcome thread dimension limitations in the GPU, and thereby provides further improvement to the processing performance.

Our approach to arbitrary resampling utilizes distinctive features of GPUs to yield a highly flexible and efficient software-based implementation. The implementation does not require a filter design process. Nor does it require complex control circuitry to calculate the fractional resampling points. The overall system is applied to 3.5G UMTS HSPA to demonstrate its capabilities as a multi-dimensional, parallel transceiver architecture for a relevant communication standard.

This work is presented in detail in Chapter 5.

1.2.4 Multirate Filtering for Multi-carrier Systems

Although arbitrary resampling is an attractive feature for various kinds of signal processing systems, it can also be useful to derive specialized resampling structures when simpler or more restricted forms of resampling are needed — e.g., to perform strictly-integer interpolation or decimation. These operations are simpler and require less hardware resources, such as multipliers and registers. In this contribution of the thesis, we explore more traditional resampling configurations, such as rational resampling, particularly using polyphase multirate filtering to process multiple channels of data.

In this work, we address the challenges of processing wide bandwidths (BWs) of data, which may be fragmented across the frequency spectrum. Modern communication systems deploy various forms of aggregation through multiple carriers. The aggregated carriers can be within the same operating frequency band (intra-band) or spread across different bands (inter-band). These issues pose significant challenges in modem design, since all of the carriers need to be present at the baseband for further processing.

We propose an efficient, rational resampling architecture that is highly optimized for real-time implementation on GPUs, and provides the foundation for effective sample rate conversion (SRC) in the context of multi-carrier transceiver system design. We propose a new SRC design method that can be applied to multiple channels and multiple carriers. This resulting architecture exhibits high throughput, low latency, and low complexity while providing the flexibility of an all-software realization. The architecture includes optimizations for significantly increasing data coalescing and occupancy in the GPU.

This work is presented in detail in Chapter 6.

1.2.5 Synchronization

Once channelization and resampling are completed, a receiver must synchronize to the incoming signals and estimate their parameters, particularly their frequency, phase, and timing [3,4]. Such synchronization is a critical aspect of receiver design. We focus on timing recovery since it is the most computationally demanding part of synchronization. Both phase and frequency recovery are performed at the symbol level, whereas timing recovery is performed at the sample level. Since there are multiple samples per symbol, timing recovery requires more data to be processed.

We present a novel GPU-based symbol timing recovery (STR) implementation that accelerates the timing error detection (TED) part of STR. TED is a critical component of STR since it is responsible for accurately measuring the timing offset. One option in TED implementation is to interpolate by a large factor to reconstruct the signal accurately, which helps to optimize the sampling time in the circuit. We adapt a feedback-based design that uses a combination of a CPU and a GPU, where the CPU handles the serial computation of the feedback, and the GPU handles the TED, which can be a fully-parallelized filtering operation. We implement a fully-parallel polyphase interpolator to speed up the computation in the GPU. Then the CPU uses the filtered results to make a decision on the optimal sampling instant. We demonstrate the achieved speedups by comparing a sequential, CPU-only implementation and our GPU-accelerated implementation.

This GPU-based STR acceleration work is presented in detail in Chapter 7.

1.3 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 provides background on various topics that are relevant for this research, including channelization, arbitrary resampling, sample rate conversion, and timing recovery. In Chapter 3, we present the notion of GPU-based front-end receiver design using an efficient GPU implementation of an arbitrary resampling polyphase channelizer. In Chapter 4, we present a high-throughput, low-latency polyphase channelizer on GPUs. In Chapter 5, we present our implementation of multi-channel arbitrary resampling on GPUs. In Chapter 6, we present an efficient GPU implementation of a multirate resampler for multi-carrier systems. In Chapter 7, we present GPU-based acceleration of symbol timing recovery. Finally, we conclude in Chapter 8 with a summary of the thesis and a discussion of directions for future work.

Chapter 2: Background Information

In this chapter, we provide background information on core concepts that are applied and built upon in the work presented in this thesis. First, we discuss the notion of a wideband channelizer, which separates channels, and then applies resampling methods, including rational and arbitrary sample rate conversion. Following resampling, a wideband channelizer must synchronize the signal to the transmitted signal. We focus in this chapter on timing synchronization in particular. Finally, we provide background on the popular GPGPU language, CUDA.

2.1 Wideband Channelization

Multi-carrier modulation (MCM) is a technique to transmit data by splitting data over multiple carriers or channels using separate carrier signals [5]. A common MCM method is to combine narrow bandwidth (BW) signals into composite signals with wider BW, and transmit these composite signals in parallel. Because of the increased BW, MCM offers increased data rate and throughput [6]. There are several forms of MCM available and a common choice is orthogonal frequency division multiplexing (OFDM) and its variants [7–9].

The underlying modulation in MCM employs the discrete Fourier transform

(DFT). In order to overcome some of the disadvantages of DFT-based modulation in MCM, a filter-based DFT modulation technique known as filter bank multi-carrier modulation (FBMC) is sometimes used [10, 11]. In order to remove noise and shape the spectrum, pulse shaping or filtering is performed prior to DFT computation. An efficient way to perform this kind of filtering is to use a bank of sub-filters along with a method known as polyphase filtering to decompose a prototype filter into a multi-rate filter [12, 13]. Such a polyphase filter bank (PFB) followed by a DFT block is an attractive option since it provides a "cleaner" MCM by reducing sidelobe leaks in the spectrum compared to an OFDM approach [6].

Even though an MCM system contains many individual sub-carriers, overall it is viewed as a single carrier system with a wider BW. Instead of servicing individual carriers separately, a front-end receiver for MCM must be able to receive and process multiple channels and carriers simultaneously. A channelizer is a communication subsystem that is used to separate multiple channels from a wide system BW. A channelization process is responsible for signal conversion (i.e. mixing), sample rate change, and filtering. A straightforward approach to channelization is to employ a bank of dedicated sub-receivers where each unit is dedicated to a single channel or carrier. However, such an approach will accumulate excessive area, power consumption, and design complexity as system requirements increase.

An attractive alternative is an all-in-one system to process multiple channels simultaneously. A type of FBMC called a polyphase channelizer is a promising option in this context. A polyphase channelizer can down convert, downsample, and filter images at the same time in a single stage [13–16]. A polyphase channelizer can be used as a synthesizer, called a polyphase synthesis channelizer (PSC), to transmit a composite of sub-channels on sub-carriers in the transmitter (TX). Conversely, a polyphase analysis channelizer (PAC) is used to separate the sub-channels in the receiver (RX) using analysis FBs [17–19]. Since the channels do not overlap in our case, we can partition the shared spectrum evenly, reduce the sampling rate, and reject adjacent channels to avoid aliasing using a PAC. Such a polyphase channelizer can be implemented using a PFB and a DFT unit to provide a flexible architecture that can be adapted to different system BWs, channel spacings, and sampling rates [13, 14].

In this thesis, we focus on developing a novel GPU implementation of a PAC for an RX system. The intensive computational requirements and stringent real-time constraints of a PAC make such a software implementation problem very challenging. However, this is an important challenge to confront given the cost and flexibility advantages of software-based transceiver (TRX) solutions, as we have motivated previously. Henceforth in this thesis, we use the terms “PAC” and “polyphase channelizer” interchangeably since our work on polyphase channelizer implementation focuses on the analysis subsystem.

A polyphase channelizer combines multiple operations into an all-in-one design. A basic operation of down channelization is as follows. A downconversion of a channel-of-interest is performed, followed by a lowpass filter to reject adjacent channels. Finally, a downsampling is necessary to reduce the sampling rate to a Nyquist rate so that unnecessary computation is avoided. A polyphase channelizer has 2 major components: PFB and DFT. A PFB is a multirate filter that performs downsampling and low pass filtering at the same time. A DFT is used to downconvert the output of PFB to baseband, and allow the user to select desired channel indices.

A prototype filter is designed that has a filter order of N. This 1D filter is reshaped, using polyphase decomposition [12, 13], into a 2D polyphase filter. A polyphase filter has Q rows or filterbanks, where each row has M columns or subfilter coefficient taps. Therefore, the overall filter length N can be viewed as having dimension of $Q \times M$. In addition, the number of DFT points equals the number of PFB rows, Q. In order to filter or perform convolution operations, a buffer is created to match the dimension of the PFB. This is an inner-product (IP) operation across each row of the PFB. A commutator is used to present the input sample in a bottom-up fashion as shown in [13,14] since this is a downsampling operation. Once all Q rows of samples have been inserted into the input buffer, the IP operation is performed.

The output of the IP produces a $Q \times 1$ vector. This vector is presented to the DFT for a downconversion operation. It is important to note that each bank or row is independent from the other rows, and as the input samples are presented one at a time, the IP operation can be performed on a per-row basis. Therefore, once the commutator reaches the top, the matrix multiplication between the input buffer and the filter coefficient matrix is complete. Since the row operations are independent of the others, they can be fully parallelized. In addition, if the input samples can be presented Q at a time instead of one sample at a time, the need for the commutator is eliminated. However, there is a trade-off in that more resources are required. For example, more multipliers are required in the commutator-free version to support parallel operations, whereas when the commutator is employed, a single multiplier can be shared across the entire matrix multiplication provided that all of the required operations can be completed prior to arrival of the next input sample.

Prior to applying the polyphase channelizer, we have an input sampling rate F_s^{in}, a given system BW, and a data rate R_d. After application of the polyphase channelizer, the input sampling rate is divided by Q and similarly, the system BW is divided by Q, yielding equally-spaced channels and a reduced sampling rate at the output of the channelizer. This is called a standard or maximally decimated polyphase channelizer [14]. In [14], an interpolation operation was combined with a maximally decimated polyphase channelizer to perform a rational resampling at the same time. This is a partially decimated polyphase channelizer. Since the interpolation in a partially decimated channelizer is being performed at some rate P while Q samples are being presented to the input, the output of the IP is presented $R = Q/P$ (R is referred to here as the rational resampling ratio) times faster than in the standard polyphase channelizer configuration. This in turn causes the need to shift the data at the input and output of the IP operation. A serpentine shift and circular shift are needed to provide such translations, which prevent phase shifts at the output of the DFT [14].

A polyphase channelizer block diagram is shown in Figure 2.1.

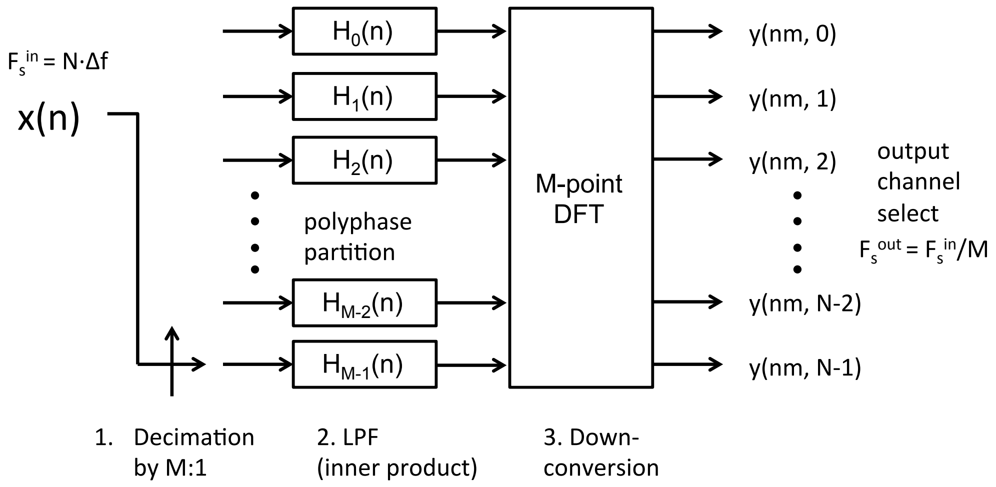

Figure 2.1: Block diagram of polyphase channelizer.

2.2 Arbitrary Resampling

Although a channelizer is capable of changing sampling rates [15, 16], it is a restricted operation, since it can only accommodate integer multiples of the output sampling rate. An additional resampler is in general necessary to resample to a desired sampling rate, ideally to any arbitrary rate. If a resampling filter is fixed to a certain ratio, then it is difficult to adapt it for other purposes. One option is to compute the associated filter coefficients on-demand based on dynamically varying filtering requirements. An asynchronous or arbitrary sample rate conversion

(ASRC) technique is commonly deployed to interface asynchronous systems to different sampling rates [20, 21]. ASRC can be viewed as a generalized rational ratio sample rate conversion approach that allows multiple kinds of conversions without requiring separate filter design processes, and allows flexible and reconfigurable conversion ratios. ASRCs can provide such reconfigurable conversion dynamically and in a single stage.

However, ASRC is a relatively complex and costly process that requires high interpolation rates or time-varying polynomial approximations. A general approach to arbitrary resampling in the discrete-time (DT) domain is to oversample or interpolate heavily to a maximum capability of the system clock so that the output sample is brought as close as possible to the desired sample index. This can be viewed as creating a dense virtual analog intermediate waveform in the DT domain [22]. This notion of creating a virtual analog waveform and resampling is known as *resample after reconstruction* [21]. Sampling in the DT domain causes images in the frequency domain at multiples of the sample rate; thus, filter design to remove images is an important aspect of SRC. The anti-aliasing filter must be carefully designed such that it rejects images appropriately, and does not distort the channel-of-interest (COI). An alternative approach would be to create a continuous time-varying signal, compute fractional resampling points on the fly, and filter out any images to control aliasing [21].

A key to SRC is in interpolation, and creating such a virtual analog waveform is equivalent to a virtual digital-to-analog conversion (DAC) process. A DAC has a zero-order hold operation, which has a rectangular response in the DT domain and a *sinc* response in the frequency domain [13, 21]. This sample-and-hold operation is essentially a nearest neighborhood (NN) interpolation. An even more accurate approximation to an analog waveform would be to linearly interpolate between two successive samples in place of NN interpolation. This first-order hold of a DAC creates a piecewise-linear approximation to the original signal in the DT domain. This type of linear (LN) interpolation can be modeled as convolving samples with a triangular pulse. This triangular pulse can be formed by convolution of two rectangular pulses in the time domain, which is the same as multiplying two *sinc* pulses in the frequency domain. This product, $sinc^2$ provides additional suppression of spectral replicas [22,23]. Higher-order interpolation, such as quadratic or cubic interpolation, is also possible, but it further increases complexity and resource usage [20, 23–25].

One option for efficient ASRC implementation is to perform high-rate interpolation initially followed by a simpler, second interpolation circuit. A second option is to use a lower level of initial interpolation followed by a second interpolation that approximates the curve between the two interpolants. The second interpolator, which approximates the curves, can be performed via polynomial curve fitting methods using linear, cubic, and spline interpolations [20, 21, 26]. A proposed architecture in [22] takes advantage of this situation by first interpolating via polyphase filters and linearly interpolating between polyphase filter outputs using triangular convolution in a single integrated stage. This architecture has performance (i.e., error resolution) that is similar to a polynomial-approximating Farrow structure filter [24, 26].

An efficient approach to interpolation by a non-integer factor is to first interpolate by an integer factor using a simple interpolator, such as a polyphase filter in the time-domain or a DFT-based *sinc* interpolation in the frequency-domain [27,28], and then apply a piecewise polynomial filter to further interpolate between samples. The computational time and error compared between a classical time-domain filter approach and a DFT-based approach are shown in [27, 29, 30]. Therefore, this combined initial interpolation with polynomial interpolation provides anti-aliasing filtering along with arbitrary resampling [23]. This represents a linear time-varying filter aspect of resampling, since each desired output sample time index would be time dependent on the input sample index and the resampling point, Δ would have to be recalculated each time.

Regardless of the method, a key to ASRC is in interpolation. The goal is not to implement a complete set of resamplers but only a fraction of them and to calculate the fractional difference, which represents the linear time-varying filter aspect of resampling [21,26]. Resampling via NN or LN interpolation alone may not be enough. An integer interpolation circuit followed by a polynomial curve fitting method will give more accurate fractional resampling points. Ideally, this ASRC should be provided in a single integrated stage with reduced complexity. Therefore, an alternate approach would be to integer interpolate first to some degree followed by a second interpolator that approximates the curve between the two interpolants. The advantage of this approach is shown in [13,22] by combining a polyphase interpolator with a piecewise polynomial filter.

2.3 Rational Resampling

A rational resampling is a combination of interpolation (reconstruction) and decimation (resampling) to achieve a targeted fractional rate. An interpolation is a combination of upsampling or insertion of zeros between samples followed by a low pass filter (LPF) to reject the resulting images. Conversely, a decimation is a combination of an LPF operation followed by discarding or downsampling of samples. By combining the two operations, one can achieve a fractional or a rational resampling ratio, $R = P/Q$, where P is the interpolation rate and Q is the decimation rate [12, 13, 31].

For resampling, the interpolation stage is critical because the sampling rate of the reconstructed signal affects the accuracy of the resampling process — higher sampling rates in general lead to higher accuracy. However, during resampling, the unwanted samples are simply discarded. In the design process, a trade-off is made so that one does not discard so much (from a signal that is too heavily interpolated) so that there is excessive computational waste, while providing a sufficient level of resampling accuracy. For example, a rational resampling approach should not be used with interpolation and decimation ratios in the hundreds. For that type of fractional resampling, an arbitrary resampler is better suited to computing the resampling points dynamically, but such an arbitrary resampling approach can add significant complexity. An alternative approach is to perform resampling in multiple stages to reduce resource usage, which in turn adds delay [12, 21, 28].

Finally, an efficient form of rational resampling exists in the form of a multirate filter using a polyphase structure [12,13]. The polyphase decomposition transforms a 1D filter into a 2D matrix, where the number of rows equals P or Q, and each column corresponds to a distinct sub-filter for the decomposition. A commutator is employed in conjunction with the polyphase decomposition to insert or select samples one at a time instead of computing zeros, which is wasteful [13].

In order to design a rational resampler, one can perform interpolation followed by decimation, or simply combine the operation into one. A filter must be designed for each interpolation and decimation. Therefore, using the same filter, one can insert the input samples using a commutator to the filter, P times then once filtered, the output can be read out every Q times, discarding $Q-1$ samples. This way only one filter is designed and simply using commutators to achieve the desired decimation rate [13]. If Q is equal to 1 then the resampling operation becomes the interpolation operation.

2.4 Symbol Timing Recovery

The timing error in maximum likelihood (ML)-based method is defined as: $t_{error}(n) = \dot{y}(n)y(n)$, where $y(n)$ is the output of the filter and $\dot{y}(n)$ is the output of the derivative filter. This equation is for low signal-to-noise ratio (SNR). However, in practice, designers apply it for all SNR [13]. This approach can be implemented using 2 polyphase filters — a polyphase matched filter (MF) and a polyphase derivative matched filter (dMF). Polyphase filter implementation is well discussed in [13,32]. This form of timing error detection has higher SNR than Gardner and faster locking average time [33]. Gardner is an approximation of $\dot{y}(n)y(n)$ using 2 samples per symbol (spS) by scaling zero-crossing of the eye diagram, therefore, it only uses a single MF.

This polyphase-filter-based ML offers an efficient option and is the most applicable to our purposes in this thesis of efficient parallelization and low complexity implementation. The number of interpolation points corresponds to the number of filterbanks. Therefore, with increasing numbers of filterbanks, we can achieve higher interpolation rates. Separate interpolation filter and MF structures result in extra processing delay. Hence, we combine the polyphase interpolator into an MF that uses 1 filter [33, 34].

This approach to ML-based TED is compute intensive, often requiring many multipliers for filtering, and filters that contain hundreds or thousands of coefficients depending on the interpolation rates and filter orders. Therefore, parallelizing and distributing the filtering tasks are attractive options and can be realized well in GPUs given their lightweight operations and data parallel processing structures. Our goal is thus to map the filtering operations across the GPU in such a way that GPU utilization is maximized, thereby offering reduced computation (from the polyphase structures employed) but faster locking and higher throughput.

If the current timing estimate is too early, then the slope of the MF output is positive, so the timing phase should be advanced to an optimum sampling point. On the other hand, if the current timing estimate is too late, the slope of MF output is negative, and the timing phase should be retarded [35]. Harris and Rice presented a modern symbol timing recovery (STR) by integrating the MF and polyphase interpolator into a single structure as a polyphase MF [32].

We employ BPSK for the simplicity and practicality of its implementation. For BPSK implementation, we follow the overall structure presented by Gardner [36] and Frerking [37]. The idea of interpolation of such digital modems is well covered in [38, 39]. We combine polyphase timing error detection (TED) [13] and ML error detectionin [32], and then we exploit the resulting algorithm structure using GPU technology to improve the performance of STR further. This method can be extended to QPSK or M-QAM for complex signals as well, and pursuing such extensions is a useful direction for future work.

Once a corrected sample point has been selected at the output of the MF, we discard the remaining interpolated samples. For example, given an input data stream at 2 samples per symbol and after 1:32 interpolation, we have 64 samples to choose from. We sample once at the peak, and then discard the remaining 63 interpolated values or interpolants. With this sampled point, we calculate the error discussed earlier. This timing error is then fed to an loop filter (LF) structure where it is used to eliminate the phase and frequency error.

Because of its rapid variation, we cannot use the instantaneous error to correct our timing. Therefore, we must average over some time, a fundamental method in signal detection. However, using averaging methods such as moving average can take too much time to lock. If the error does not change, then the system remains locked, but unfortunately timing jitters due to noise may persist, and these errors must be detected and corrected (filtered) each time. An LF is useful to provide such correction. Following the LF, we need an NCO to correctly drive or adjust

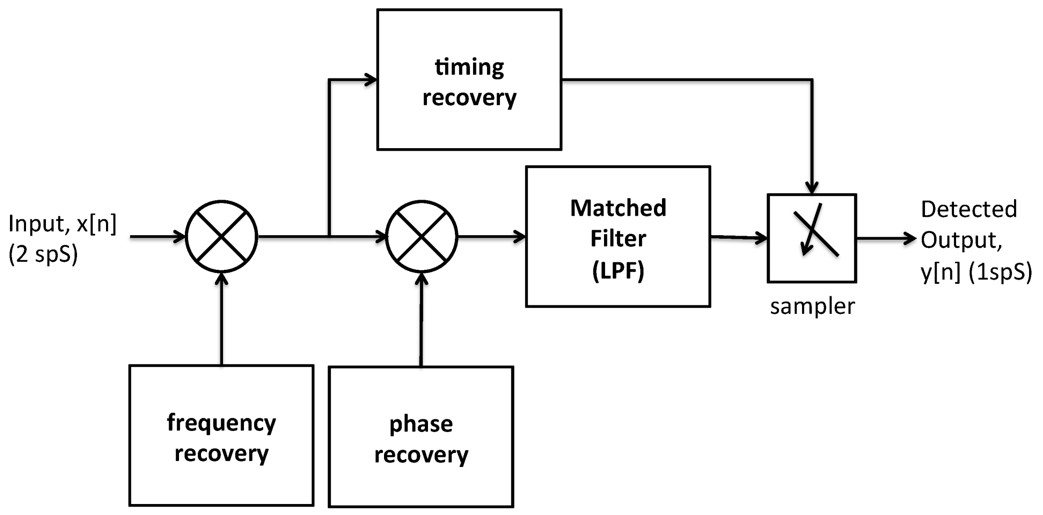

Figure 2.2: Block diagram of different synchronizers.

the timing sample and feed it back to TED for future correction or adjustment as necessary. A feedback control circuit is used to accomplish this task by selecting a corresponding filterbank index.

Figure 2.2 shows a block diagram of our targeted communication receiver synchronization including frequency, phase, and timing recovery subsystems. We assume that we have a single carrier narrow band system at the baseband.

2.5 CUDA

NVIDIA introduced Compute Unified Device Architecture (CUDA) [2] as a parallel programming language for programming GPUs for use in graphics, as well as in other computationally intensive application areas. CUDA is based on a single instruction multiple thread (SIMT) programming model, where multiple threads execute the same instructions over different data sets. The SIMT model provides an attractive model for implementation of SP algorithms.

A CUDA kernel is a grid set of blocks, and a block is a set of threads. A processing core is referred to as a streaming multiprocessor (SMP). A group of 32 threads is called a *warp* and is executed as a group inside an SMP. The total number of threads inside a block should be multiples of a warp. To achieve maximum performance, it is important to keep the GPU as busy as possible, and utilize as many threads and blocks as possible.

The GPU memory hierarchy also needs to be utilized carefully to maximize performance. An external memory or global memory (GM) is the largest memory in a GPU system, and is also the slowest memory. GM is commonly used to transfer data back and forth between the GPU and a corresponding host CPU. Shared memory (SM) is contained within an SMP and is visible only to a specific block of threads. It is a fast read/write local memory that can be viewed as a fast, user-enabled cache. A constant memory (CM) is a fast, read-only memory for storing constants. In addition, there are registers for local variables. Any register spills or arrays that are not in SM are stored in local memory (LM). It is important to utilize

SM as much as possible since registers are limited, and GM and LM are relatively slow.

There are several important features of CUDA that require careful attention when programming GPUs. First, memory transfers between CPU and GPU must be minimized due to the high latency of transferring data over the bus. Accesses to GM should be coalesced whenever possible and SM should be used to avoid unnecessary access to GM. Grouping threads in multiples of a warp facilitates coalescing of data with GM and helps to enhance GPU utilization. Finally, computational work should be spread throughout the GPU to utilize more cores and enforce lightweight operations. The programmer typically profiles the application extensively to fine tune performance and identify bottlenecks. To summarize, a GPU programmer should design kernels to spread the workload as much as possible throughout the GPU, read from GM in a coalesced manner to an SM, instantiate sufficient numbers of threads per block (TPB), and write back results to GM in a coalesced manner. For more details on CUDA programming, we refer the reader to [2, 40].

2.6 Summary

In this chapter, we have provided general background information and reviewed various concepts and methods that will be applied in the following chapters of this thesis. We have presented an overview of the overall structure of our GPU-based, front-end receiver architecture. This architecture includes capabilities for channelization, different forms of resampling, and timing synchronization. Finally, we have presented an overview of CUDA, which we use as a tool map our proposed front-end receiver structures into an efficient wideband GPU front-end receiver implementation.

Chapter 3: A GPU Implementation of a Front-end Receiver

In this chapter, we introduce the notion of a GPU front-end (GFE) receiver architecture using channelization followed by a resampling method to down convert, filter or reject unwanted channels, and properly resample to the desired sample rate for all of the channels simultaneously. We implement an efficient polyphase channelizer architecture using interpolating channelizer and time-varying resampling methods. We develop optimized implementations of these methods that operate entirely in a GPU to implement our parallel GFE transceiver. We demonstrate the performance of our design by applying it to the popular 2G wireless standard, GSM. The work presented in this chapter was published in [41].

3.1 Introduction and Related Work

A communication receiver can be divided into 2 major systems: a front-end and a back-end system. A front-end system is responsible for channel estimation, down conversion, and sampling rate change. A back-end system is responsible for channel and source decoding. This notion is well established in [3] using an inner-receiver and outer-receiver, respectively. Advancements have been made over the years to implement a majority of these functionalities from intermediate frequency (IF) to baseband signal processing (SP) in software. The goal of the software-based receivers is simple — to bring the SP functionalities closer to the antenna as much as possible, reducing the burden on the RF front-end (RFFE), and exploiting the flexibility of software and the processing power of modern hardware. Dedicated hardware devices, such as ASIC/ASIP/FPGA devices, have made substantial advancements over the decades and along with the focus on low-power multi-core microprocessors, we now have additional platforms, such as GPUs available to use.

In order to effectively utilize GPUs, a general purpose GPU (GPGPU) programming language such as CUDA [42] can take advantage of a GPU's many lightweight threads and many cores to perform complex SP functions in parallel by exploiting and exposing data parallelism commonly found in SP problems. Our goal in this chapter is simple: use a GPU as a front-end receiver and bring it as close to the antenna as possible, thus accelerating and improving performance significantly as a GPU front-end (GFE) receiver. Such a system can co-exist and cooperate with existing hardware or ideally use a GFE as a stand- alone unit (replacing existing hardware based modems). We aim to use a GFE at least in conjunction with existing hardware. To provide for flexibility, we seek to perform minimal processing on hardware (e.g., only where dedicated acceleration is essential to meet performance constraints), and have maximal functionality operating on software-based GPUs.

An analog front-end (AFE) is primarily responsible for down converting the entire frequency bandwidth (BW) centered at some much higher RF frequency to some IF that is usually running at a quarter of the sampling rate of the ADC. Within the large AFE input BW lie multiple narrower BW channels, including the channel-of-interest (COI). The main task of the digital front-end (DFE) is to down convert the COI to the baseband for further processing [3, 13, 43]. In addition to this conversion, a filter must be used to reject adjacent channels to select the COI. Following this filtering, the sampling rate needs to be reduced since it is usually sampled at tens or hundreds of samples per second out of an ADC. However, all of these tasks are strongly coupled, and there is a strong relationship between down conversion and filtering, which are the main tasks of a channelizer along with sampling rate conversion [21]. Therefore, a DFE is primarily responsible for two tasks: channelization and sampling rate change, which are necessary for most air interfaces of wireless communication systems.

Channelization is a process of separating multiple users or channels commonly found in FDM schemes, i.e. each channel occupies its own BW given a much wider system BW and many channels within it. This is a common way to share a radio spectrum and is found in application domains that include radio astronomy, broadcast TV and radio, etc. There are 3 basic tasks for a channelizer: 1. down conversion, 2. downsampling, and 3. rejection of adjacent channels via filtering. These tasks can be accomplished independently or jointly. A modern channelizer uses multirate SP techniques to achieve different tasks at the same time, such as using polyphase filter bank (PFB) techniques introduced in [14]. This method allows all-in-one solutions using inner- product (IP) and discrete Fourier transform (DFT) operations. The input to such a system is a frequency domain multiplexed (FDM) signal, and the output is a time domain multiplexed (TDM) signal, where each channel corresponds to an index of a DFT. This makes the polyphase channelizer a versatile system for a front-end receiver.

For baseband processing, manufacturers prefer fixed clock rates over tunable clocks due to their advantages in terms of cost, accuracy, and stability. Sampling rate conversion (SRC) is generally necessary between a fixed clock and some desired sampling rate. Through such use of SRC, many different data rates can be accommodated from a common fixed clock. Often, non-integer resampling, such as fractional resampling, is needed [13, 21]. In the development of such resamplers, polyphase filters are of particular interest due to their ability to reduce resources, perform filtering, and convert sample rates simultaneously. In addition, PFB decomposition provides matrix structures that can be used to perform efficient IP operations. Using GPUs, we can implement the associated IP operations in parallel using all of the available threads and resources [44].

We address the limitations of polyphase channelizer by eliminating additional buffers and complex multi-dimensional buffer management using a simple indexing scheme on a single-dimensional array. We introduce the notion of assigning a channel to a block of threads in the GPU, allowing multiple channels to be processed in parallel across multiple blocks, thereby enabling scalable, high-throughput parallel receiver. We demonstrate efficient GPU-based techniques to achieve arbitrary resampling. Finally, we integrate our methods for performing polyphase channelizer, SRC, and parallel, multi-channel processing to derive a powerful GFE receiver design, and we discuss performance results from our prototype implementation of this receiver.

3.2 Polyphase Channelizer

A typical base station (BS) contains N sets of sub-receivers to down convert and demodulate multiple narrowband RF channels. Further, each sub-receiver has its own RF mixer that will down convert from some carrier frequency to an IF, followed by an additional task of down converting to a baseband. This task of having one dedicated sub-receiver per channel and processing the channels independently can be a daunting task, often requiring a "room full of receivers", not to mention excessive power consumption. This per-channel approach is used because of its flexibilities, but it requires a large amount of resources. Instead of building N one-channel or per-channel receivers in parallel, one can build a single receiver to process all the channels at once [43].

Although the idea of channelization has been around for a long time, a modern channelizer was introduced in [14] using polyphase filters that perform multiple tasks simultaneously. Using a PFB means having a parallel arrangement of filters that is responsible for different outputs of the spectrum of the signal. The polyphase filter as a multirate filter performs simultaneous sampling rate change and lowpass filtering. In addition, using a DFT as modulation can provide a more cost effective solution than N parallel independent filters and mixers [43]. A basic operation of channelization is as follows: a down conversion is necessary to bring COI to baseband, a lowpass filter is used to remove the adjacent channels to reject co-channel interference, and finally, a downsampling is needed to reduce an unnecessarily high sampling rate and the associated computational burden. Following this channelization, an SRC may be necessary to convert the sampling rate to desired multiples of the data rate.

An alternate solution to performing channelization as a single merged process using a PFB structure offers major advantages by reducing resources while processing multiple channels at the same time. We will not discuss all the details of transforming bandpass filters into a channelizer here [13], instead we will provide an overview. We will refer to a channelizer as a polyphase channelizer from now on.

In a standard polyphase channelizer configuration, one can simply divide the input sample rate by the number of filter banks, Q to derive the output sample rate (i.e., downsampling by an integer factor). Similarly, the system BW is also divided by Q into equal-sized channel BWs. The number of PFBs matches the number of DFT points. This is called a maximally decimated polyphase channelizer [14], where the downsampling rate equals the number of PFBs. A notable contribution in [14] introduces an arbitrary bandwidth, channel spacing, and output sample rate channelization by performing interpolation at the same time simply by partially decimating and rearranging the data between the filter and DFT stages. This is effectively a rational resampling channelizer. We distinguish these two types of channelizers by referring to a standard channelizer as a maximally decimated channelizer, and the other (being the interpolating channelizer) as a partially decimated channelizer [14]. A polyphase channelizer is shown in Figure 3.1 along with an optional SRC stage at the end to represent a basic DFE block diagram.

Prior to channelization, a conventional filter must be decomposed into a 2D polyphase filter matrix [12]. An input buffer must be prepared to perform an IP

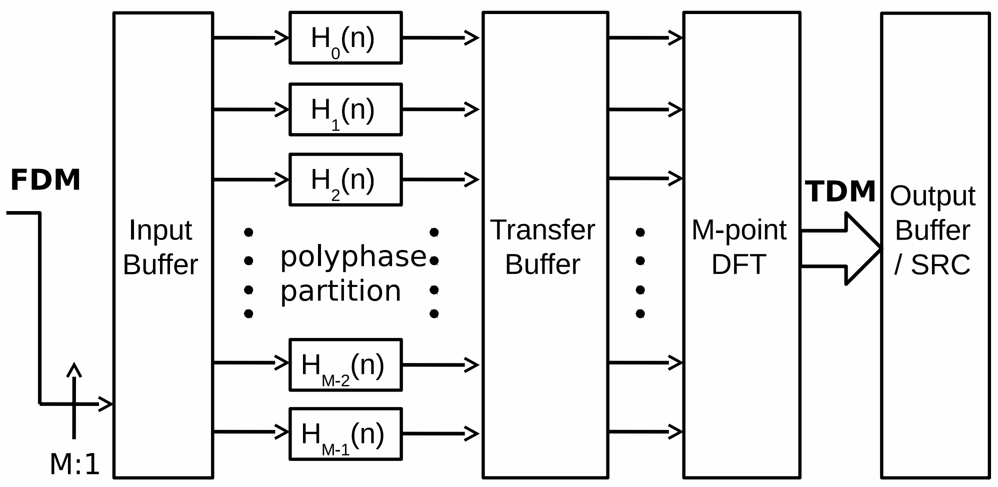

Figure 3.1: Block diagram of polyphase channelizer implementation.

between the input samples and the filter coefficients to perform a convolution operation. An additional input buffer or a shift register is used to accomplish such a task. A polyphase channelizer uses an input buffer, a filter coefficient matrix, a transfer buffer, and a DFT to perform maximally decimated channelization, i.e. $F_s^{out} = F_s^{in}/Q$, where Q is the number of PFBs, which is also the number of DFT points. In this scheme, the input buffer and filter matrix have an identical dimension of $Q \times M$, where Q is the downsampling rate or number of rows and M is the length of the subfilter or number of columns. The output of the IP produces a $Q \times 1$ vector. Since a commutator is used to present the input one sample at a time to the input buffer, a transfer or an intermediate buffer is used to store all the computed IP values prior to the DFT. Finally, as seen in Figure 3.1, an output buffer can be installed optionally to store all the data following the DFT. Therefore, it is necessary to use shift registers or buffers at every stage to compute the IP and DFT in a polyphase channelizer.

For the partially decimated polyphase channelizer presented in [14], the complexity increases significantly, especially with managing the buffers. Overall, the IP computation and DFT stay the same but the buffers have to be manipulated to absorb all of the phase shift in the time domain. Reference [14] presents a serpentine shift to move around the input samples for the IP operation. Because it is interpolating at some rate of P in addition to decimating at the rate of Q, the IP is being produced P times faster than the standard polyphase channelizer. Since the IP matrix dimension is the same as before, this creates overhead to manipulate the buffer to prepare for the next set of Q/P or R input samples. For a hardware device such as an FPGA, this is realized by using a system clock that is operating much faster than the input sample rate. This is still a cumbersome task to perform in the hardware. Given an input stream arriving serially, one would have to create a 2D shift register to perform the serpentine shift. In the meantime, the remaining rows of banks must continue to compute the IPs, and therefore, additional pointers are needed to keep track of all the IPs to be computed in timely manner for a DFT operation. In fact, one would need P pointers, which introduces additional overhead in implementation.

Following the IP operation, an intermediate buffer is used to collect all the

IPs one at a time. However, because only R samples out of Q samples are the new IPs, a phase shift occurs at the output of the DFT. This is because the remaining samples are the time delayed version of the previous set of IP values. Therefore, the data must be circularly shifted at the input of the DFT in order to absorb the phase shift at the output of the DFT [14]. Without this circular shift, the output of the DFT will exhibit a sign reversal on every other DFT output.

Therefore, theoretically speaking, a rational resampling in this channelizer is possible. However, in practice this does not appear to be practical using current technology. The examples presented in [14] use interpolation rates that are integer multiples (2 or 4) of the data rate, since discrete domain systems have a close relationship between sampling rate and data rate that are multiples of each other. This is limited if an all-discrete solution is applied to an analog or a non-integer multiple discrete systems. In addition, serpentine shifts and circular shifts can be performed easily if they are in multiples of even numbers. For an odd integer multiple, such an operation is more complex and may require additional buffer overhead. This is a trade-off of using an all-in-one channelizer, as in a polyphase channelizer — while simple in architecture, its operation is complex and has no flexibility.

Using extra buffers between successive stages simplifies the implementation but increases overhead and resources. For a system such as a CPU or GPU, the data is already stored in an array, whereas in hardware, the input data is streaming in, and one needs to buffer the samples as they arrive. In the following sections, we address these issues by essentially eliminating the serpentine shift operation and eliminating the commutator using grouped memory access. We simplify the architecture considerably and make it more flexible by utilizing the GPU's many core architecture and memory hierarchy to process multiple channels simultaneously. This makes our channelizer design readily adaptable across different applications.

3.3 Arbitrary Sample Rate Conversion

A sampling rate conversion (SRC) or resampling process is a technique to convert from a fixed sampling rate to another fixed rate that is operating at a different clock frequency, data rate, etc. A sampling rate can be increased, decreased, or both at the same time using upsampling, downsampling, or rational resampling, respectively. This rational resampling process is a two step process that can be characterized as a ratio, $R = P/Q$, where P is the upsampling rate and Q is the downsampling rate. An upsampler can be realized by inserting $(P-1)$ zeros between successive pairs of original samples. Conversely, a downsampler can be realized by discarding $(Q - 1)$ samples from each block of Q original samples. Along with this sample rate change, an anti-aliasing filter is necessary to reject images. This anti-aliasing filter is the most prominent constraint to be obeyed by any SRC system [21].

An integer-factor resampling can be achieved by setting P or Q to 1. A straightforward approach is to use a cascade of these integer ratio filters to obtain some fractional sampling rate of R. However, in practice, P needs to be large enough to compute all possible interpolated points or interpolants, which are then discarded at the rate of Q samples. This is computationally wasteful. One can use a single fractional resampler or a cascade of an upsampler, a filter, and a downsampler to achieve fractional resampling. The former can make the filter very complex but runs at a lower rate, whereas the latter simplifies the filter implementation but runs at a higher clock rate [21]. More efficient realizations can be achieved using a polyphase filter approach, which combines these groups of operations in an optimized way [13]. However, polyphase realizations are limited in that they can only achieve rational resampling through indexing of PFBs. Although this is relatively simple to implement, it leads to redundancy. It is also inflexible since the sample rate change is dependent on a precomputed filter that is designed for a certain fractional rate only. In addition to rational resampling ratio, the ratio between the sample rate and the bandwidth (of the COI) is very important [43].

An arbitrary SRC is an integer-ratio oversampling based technique that allows multiple conversion ratios without requiring separate filter designs. It is based on the idea of resample after reconstruction [21]. It can arbitrarily interface asynchronous systems to different sampling clocks [20]. In many applications, an SRC is needed to arbitrarily resample the fixed F_s coming out of an ADC or DAC, but in this chapter, we resample after the channelizer to meet the data rate required for the baseband processing. In polyphase filter based resampling, the filter coefficients are precomputed and we design a P-path filter accessed by Q indexing to implement a P/Q resampling operation. Interpolation is important for SRC to be effective. Ideally, P should be large enough so that the output sample is as close as possible to the desired sample position. This is called nearest neighbor interpolation. However, often times, the desired sample point lies between the two available interpolants.

One can derive such an intermediate point using linear interpolation [13].

In linear interpolation, we first interpolate to the maximum output sample rate, essentially forming a "virtual analog signal" from the resulting samples, analogous to a DAC. Then we resample this virtual analog waveform at the desired sample locations [13, 22]. We use a polyphase interpolator to obtain the available interpolants, effectively achieving nearest neighbor interpolation. However, this becomes inefficient as we try to achieve a higher interpolation rate (finer granularity), i.e., as P increases. This is because an increase in P leads to an increase in the filter order, which in turn increases memory requirements. It is desirable to interpolate to a sufficient degree such that further interpolation can be performed linearly. Linear interpolation is achieved by convolving the input samples with a triangular pulse. In the frequency domain, a triangular pulse is the product of transforms of two identical rectangular pulses [13, 22]. The linearly interpolated sample between 2 samples at k and $(k+1)$ can be expressed as:

$$x(k+\Delta) = x(k) + [x(k+1) - x(k)]\Delta = x(k) + \dot{x}(k)\Delta \tag{3.1}$$

where k is the PFB index, and Δ is the time offset between the interpolant and the desired sample point. We now have a pair of polyphase filters that compute the interpolant and the derivative of the interpolant located at $n + k/P$, where n is the input sample index. We then interchange the interpolation operation between the output samples and the filter coefficients. Finally, using the Taylor series expansion, we obtain the following expression for the desired output:

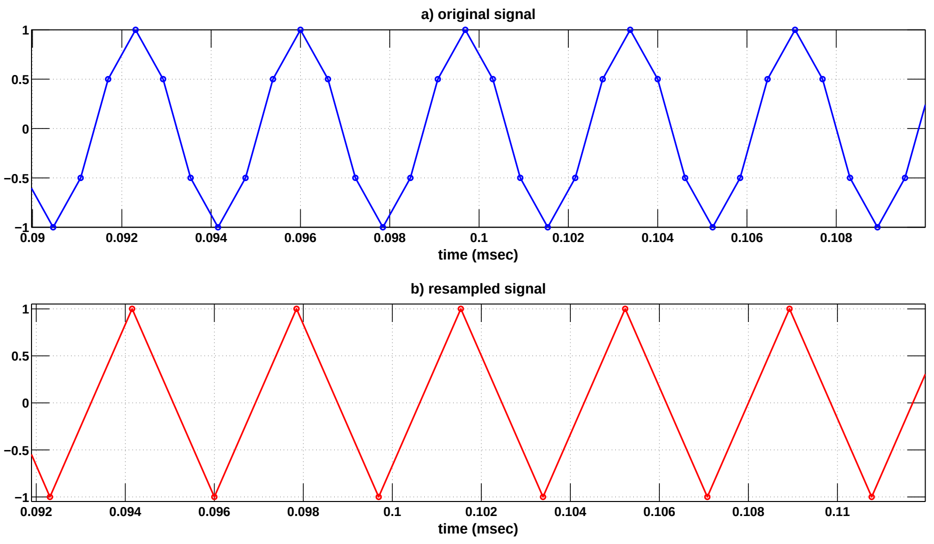

Figure 3.2: Plot of polyphase arbitrary resampler. The top figure shows a sinusoidal wave at 270.833 kHz sampled at 1,625 kHz. The bottom figure shows the same 270.833 kHz signal resampled to 541.667 kHz.

$$y(n+\frac{k+\Delta}{P}) = y(n+\frac{k}{P}) + \dot{y}(n+\frac{k}{P})\Delta \tag{3.2}$$

We are now ready to arbitrarily convert the sampling rate. The resampling error needs to be kept less than the quantization error of the input signal [13, 22]. Since we are using floating-point values, this timing jitter error is kept very low. This is achieved using a phase accumulator to track the time difference between samples. The accumulator can be sped up or slowed down depending on how fast it is incrementing, much like a time varying or continuous time resampler [20, 22]. An example plot of using a polyphase arbitrary resampler is shown in Figure 3.2.

3.4 GPU-based Arbitrary Resampling Polyphase Channelizer

We map our targeted algorithms onto a GPU to strategically take advantage of the compute resources given the parallelism in the application. We implement the polyphase channelizer on the GPU using an IP kernel and a CUFFT kernel. CUFFT is a highly optimized standard FFT library component that NVIDIA provides. We process real I-Q values instead of complex values in the GPU.

Previously, we discussed the operation of maximally decimated polyphase channelizer and partially decimated polyphase channelizer structures. We have shown the benefits of computing IPs in GPUs [44]. We apply these techniques to perform a fully parallel version of IP computation to optimize throughput. A major difficulty of performing channelization is presenting the input data for IP computation. In [14], an input buffer is used to arrange the data in a 2D format so that IPs can be performed for all $Q \times M$ values. However, in our implementation, we eliminate such a buffer by indexing through the incoming input stream, which is presented to the GPU as a single-dimensional array. For polyphase filtering, we load values columnwise, but we compute across matrix rows. In addition, due to downsampling, the input samples are loaded in a bottom-up manner using the commutator approach. Using the GPU, we can utilize coalesced memory transfers by reading in all Q input samples onto an SM once, and indexing through them using the optimized indexing scheme illustrated in Figure 3.3. This eliminates the additional buffer required to perform 2D shift registering, and any additional overhead to move the data around. These techniques allow for a fully parallel IP computation on the targeted GPU.

For the partially decimated polyphase channelizer, we perform a similar task as the maximally decimated polyphase channelizer, except that the data is only read R samples at a time, where $R = Q/P$, as discussed earlier. Ideally P should be a multiple of 2 to facilitate efficient data swapping. For example, for $P = 2$, the buffer is filled from the half way point to the end; for $P = 4$, it is filled from one quarter of the distance into the buffer; and so on. In the meantime, P pointers are employed to perform the IP operations on the remaining rows. If P is an odd number, the input buffer management becomes more complicated with serpentine shifts and increased buffer cost to handle all of the data swaps resulting from fractional resampling.

Now using the GPU, we access a block of R samples at once from the input stream, and load the block into SM to perform IPs across all of the filter banks at the same time. We do not need a commutator to read input samples one at a time any more, instead we perform vector processing. Similarly, we do not need any additional input buffers, nor an additional P data pointers to compute the IPs, and finally, we do not need to perform the complex serpentine shift. The buffer management is handled through an indexing scheme, as illustrated in Figure 3.3. This indexing scheme can be adapted to hardware as well to use of a 1D array in memory without forming a 2D serpentine shift register, and the scheme can support arbitrary values of P. Using this approach, we effectively simplify all of the operations to be based on a linear input array, and therefore, do not have to incur costs associated with difficult multi-dimensional indexing and additional buffers. This helps significantly to streamline the overall channelization process.

For the partially decimated polyphase channelizer, only R input values are presented although a total of Q IP values are computed on each CUFFT invocation. The remaining $(Q - R)$ values are IP values between previous input samples with different sets of filter coefficients. This causes a phase shift in the DFT and must be absorbed at the input of the DFT. A simple solution is to employ a finite state machine (FSM) that switches between states whenever R samples are read. There is a total of P states [14]. This is a critical operation, and in a GPU, this process is made simpler by using coalesced writes to GM. Such coalesced writes can be employed when IPs are performed, and data can be swapped depending on the state transitions. All of these improvements further simplify our implementation, and allow for improved efficiency due to coalesced memory access, elimination of buffers, and parallelism exposed using a GPU. We capture the overall process in the pseudocode shown in Figure 3.3.

We have, through our developments up to this point in this section, presented a complete GPU-based implementation of polyphase channelizer. We have also incorporated an optional feature that allows integrated processing for outputs of the Q-point DFT. In a serial implementation, such as in a CPU, we can perform the DFT and SRC at the same time by "rolling in" the SRC functionality with the DFT. If it is necessary to process all channels, then all of the DFT outputs can be buffered, and then processed individually using an iterative process. In a hardware implementation, this overhead can be mitigated by processing each channel very fast using a faster clock source; however, this may lead to unwanted power consumption and heat dissipation.

```
for $nn = 0$ to $INPUT\_LENGTH - 1$ do
    $SM\_input[ix] = GM\_input[nn + (ix \times Q + iy)]$
    SYNC
    $SM\_prod[ix] = h[ix \times Q + iy] \times SM\_input[ix]$
    SYNC
    $IP+ = SM\_prod[ix]$
    if $state = 0$ then
        write to GM
        go to next state
    else if $state = 1$ then
        swap the data then write to GM
        go to next state
        ...
    else if $state = P - 1$ then
        swap the data, then write to GM
        return to $state$ 0
    end if
    $nn+ = R$
end for
```

Figure 3.3: Pseudocode for channelizer operation. Here, ix is the thread index, and iy is the block index.

We use separate kernels to perform further processing after CUFFT. The final output of CUFFT resides in the GM and we cannot simply roll the SRC into the CUFFT since we do not have access to the CUFFT kernel. However, the channelization is now complete and CUFFT provides the system with all of the TDM outputs at once. In contrast to a typical hardware implementation, outputs from the DFT will arrive serially.

We can now perform SRC on all of the channels simultaneously. We perform SRC using the polyphase arbitrary resampler we have presented earlier in this chapter. Using the GPU, we perform nearest neighbor interpolation followed by linear interpolation to reduce timing jitter errors. Similar to our channelizer implementation, we utilize SM to perform IP computations as well as computation of fractional differences. We spread the workload across the threads in an effort to maximize exploitation of parallelism. A corresponding pseudocode description is shown in Figure 3.4.

Dedicated VLSI implementations of polyphase channelizers have been demonstrated in prior work (e.g., see [34,45–47]). The novelty of our work in this chapter is in our complete *GPU-based* polyphase channelizer and SRC implementations, and our associated methods for parallel processing of the channels. Our efficient GPU-based implementations of these important wireless communication subsystems help to significantly reduce costs and development time and increase flexibility compared to dedicated VLSI implementations. The result is a powerful, cost-effective GFE receiver that performs all of our targeted tasks and achieves our goal of integrating GPU processing "closer to the antenna."

```
**for** $nn = 0$ **to** $INPUT\_LENGTH - 1$ **do**
  $SM\_reg[ix] = GM\_input[(nn + ix) \times Q + iy]$
  SYNC
  **while** $accum < P$ **do**
    compute PFB index, $k$
    compute fractional difference, $\Delta$
    $SM\_prod[ix] = h[ix \times P + k] \times SM\_reg[ix]$
    $SM\_dprod[ix] = dh[ix \times P + k] \times SM\_reg[ix]$
    SYNC
    $IP+ = SM\_prod[ix]$
    $dIP+ = SM\_dprod[ix]$
    $output = IP + \Delta \times dIP$
    $accum+ = \Delta$
  **end while**
  $nn + +$
**end for**
```

Figure 3.4: Pseudocode for polyphase arbitrary resampler.

We have demonstrated a novel form of inter-channel parallel processing, where different channels are mapped to different blocks of GPU threads, and are processed in parallel to provide linear (in the number of channels) increase in communication system throughput. Our implementation demonstrates that a sufficient number of threads is available within such a block to perform the necessary filtering operations and other relevant computations. In contrast to implementing N channels in parallel using N separate hardware receivers, our solution demonstrates the processing of a large number of channels independently within a single GPU, thereby facilitating green computing with low power consumption and resource requirements. This is attractive for both base stations, where resource usage is critical, and mobile stations, where power consumption is critical.

In the remainder of this chapter, we focus on demonstrating and experimentally evaluating the performance of our novel wireless communication subsystem designs, and our proposed approach for GPU-driven, multi-channel processing.

3.5 Implementation and Experimental Setup

For our design and implementation, we consider GSM [48], a popular second generation (2G) wireless communication standard, particularly as a base station (BS) to process multiple uplink channels from mobile stations (MSs) all at once. GSM has 124 channels in 25 MHz system BW with channel spacing of 200 kHz, and data rate of 270.833 kHz using GMSK modulation. Many digital communication systems use data rates that are related to the sampling rate, but for nonlinear modulation such as GMSK, they can be unrelated and at a fractional rate instead of integer multiples of the data rate. This limits using a typical polyphase channelizer due to strong coupling between the data rate, channel spacing, and sampling rate. Therefore, fractional SRC must be accompanied to get to the correct integer multiples of the desired data rate afterward.

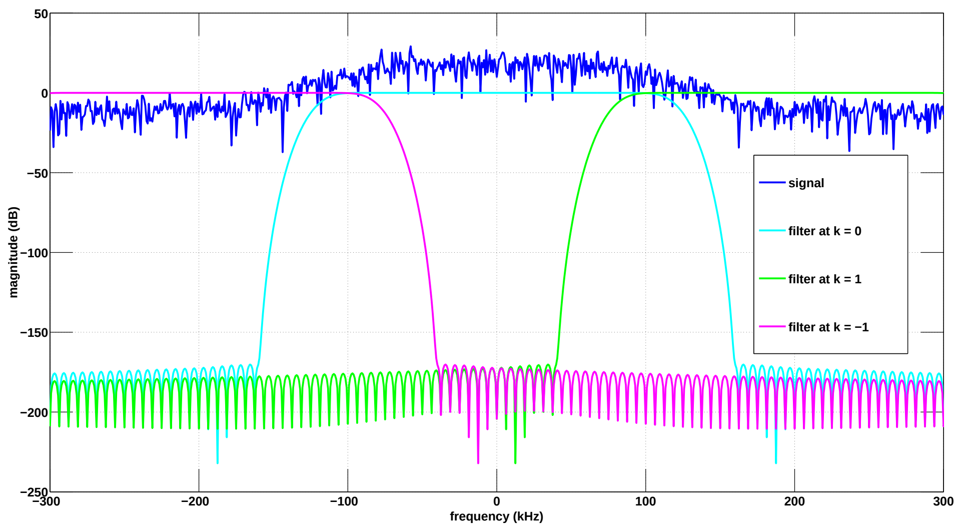

Figure 3.5: Plot of GMSK modulation with overlapping polyphase channelizer filters spaced apart at 200 kHz.

First, we design our filter to be used in the polyphase channelizer, using an equiripple FIR filter with order of 4,096, and we design the channelizer as a high-quality spectrum analyzer since there is no guard band between channels. However, due to GMSK signaling [4], most of the energy is concentrated in the center of the channel, therefore, we allow crossover BW between the channels. Thus, the filters overlap in the transition region, as shown in Figure 3.5. We use our FIR filter as a single GMSK spectral mask as long as we can mask out the COI.

Since there are 124 channels to process, we perform a 128-point DFT in the channelizer using CUFFT. The number of DFT points is also the decimation rate. Therefore, we have a $Q \times M$ IP matrix, where $Q = 128$, which is also the number of PFBs, and $M = 32$, which is the subfilter length, and is also the size of a warp in CUDA. In other words, for every Q input samples, we have Q IP values as input to a Q-point DFT, which in turn outputs Q channels. This configuration gives us a maximally decimated channelizer.

However, in case of GSM, where the channel spacing is 200 kHz and the data rate is 270.833 kHz, it is challenging to reduce the BW and F_s to satisfy both criteria after channelization. Since the data is complex, it must be sampled at least the data rate to avoid aliasing. Therefore, a standard maximally decimated channelizer cannot be used for an application such as this, where the data rate is greater than the channel spacing or when F_s is not an integer multiple of the channel spacing or data rate. The channelizer must be modified to increase F_s.

Previously in this chapter, we presented a partially decimated channelizer, which increases the channelizer's output sampling rate by interpolation. We generally cannot change other parameters without affecting the entire system. Therefore, we partially decimate or interpolate by some value P that yields F_s that is greater than the data rate. For simplicity, we use $P = 2$ to give $F_s = 400$ kHz. This clearly satisfies the Nyquist criteria; however, typically, we need F_s to be 2–4 times the data rate for proper baseband processing, and therefore, a higher degree of interpolation may be needed.

In GSM, we cannot fractionally resample using the channelizer alone simply by using the input and transition buffer we have discussed earlier. This can be a daunting task and is a prime reason why polyphase channelizer usage is limited. Even if we are able to resample at multiples of the data rate, we would need, using conventional techniques, a complicated filter design with high order filter coefficients, word lengths, extra buffers, overheads to manage complex shift register operations, and possibly, a higher clock rate, which are all expensive using direct implementation. Therefore, an additional SRC is necessary to accommodate such a fractional data rate, and must be applied to all channels.

In Section 3.4, we presented a polyphase interpolator based arbitrary resampler to dynamically compute the fractional difference of the output and desired sample points. We design the associated filter to be an equiripple FIR filter with a 32 P-paths filter. We also presented an approach to assigning a channel to a dedicated block inside the GPU, enabling parallel processing across all channels using a single GPU. This leads to a complete solution to achieving any desired sampling rate, channel spacing, and down conversion in a fully parallel structure to achieve high throughput. Furthermore, all of this is done in a (GPU-based) software framework to enable great flexibility in terms of algorithm and application experimentation and modifications. Using the algorithms we presented previously, we can now resample from 400 kHz down to the desired data rate of 270.833 kHz on all channels.

In our implementation, there are 3 kernel calls in the GPU: *1. IP compute*, *2. CUFFT*, and *3. SRC*. For IP compute, we do not create any buffers, but rather use a single dimensional input array as is, using the indexing scheme shown in Figure 3.3. We assign each PFB to a block for computing the IP, which is performed using registers and SM. The IP compute kernel is optimized further by placing the filter coefficients in CM for fast read-only broadcast to multiple blocks. At the output of the IP, we use a 2-state FSM to switch between swap and no-swap for presenting the data to CUFFT. The output of the CUFFT is then used to compute SRC for every channel. Each channel is assigned to a block to change the sampling rate across all TDM channels simultaneously. The separate kernel calls are used to reshape the block and thread dimensions each time, which is achieved by exploiting the flexibility of our software-based implementation.

Using our approach to design a GFE receiver for a BS, the overall operation of channelization can be reversed to transmit multiple channels at the same time. This is called a transmultiplexer [14]. A BS would apply a transmultiplexer in the downlink when it broadcasts to multiple MSs. If our GFE scheme is to be implemented in MS using a mobile GPU, instead of using banks of bandpass filters to select an active channel, a simple selection of DFT index can be employed since all of the channels are available for use at the same time. In addition, our implementation can readily support the frequency hopping option of GSM by simply selecting different DFT indices and avoiding additional overhead. These are some of the distinct advantages of our GFE-based implementation over conventional hardware designs, in addition to the advantages gained by using a software-based approach.

Our implementation can be adapted to other modulation schemes, such as linear modulation, and also to other applications, such as FM radio and TV broadcast. Exploring and optimizing this adaptation to other modulation schemes and applications is a useful direction for future work.

3.6 Results and Analysis

GSM traffic can be broken into two type of channels: traffic and control channels. We focus our attention on the traffic channel (TCH) only. Within TCH, the GSM frame structure uses slots, frames, and multiframes to communicate between the BS and MS. There are 8 time slots per GSM frame. Each time slot covers a duration of 0.577 ms, and thus, each frame lasts for 4.615 ms. A GSM multiframe (MF) is a basic unit of measure in this context. Each traffic MF consists of 26 frames, with a duration of 120 ms. On the other hand, a control MF contains 51 frames. We focus on traffic MFs for our analysis since most of the overall communication traffic comes from traffic MFs.

Figure 3.6 shows a plot of over-the-air GSM signal containing a broadcast channel in the center and a traffic channel (e.g. channel 6). Figure 3.7 and Figure 3.8 show the broadcast channel and traffic channel after channelization, respectively. We observe that the broadcast channel is always on as it should, since it is broadcasting the cell information all the time, and it also has more power. The traffic channel is bursty since it is an FDM and a TDM signal, and only contains the necessary data given its channel frequency and time slot.

We used NVIDIA's GeForce GTX 680 as the target GPU platform to implement our design. The GTX 680 has 2 GB of GM, 1,536 CUDA cores, 8 MPs, 64 kB of CM, and 48 kB of SM. We used CUDA driver version 5.0, and a compute capability of 3.0. Although our focus is on throughput, we have summarized our results for both runtime (the latency for processing a single MF) and throughput in

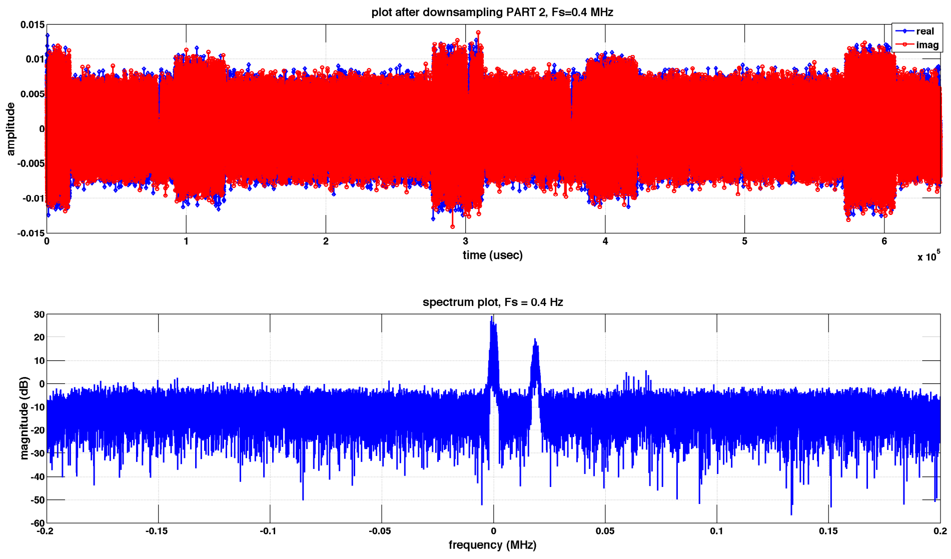

Figure 3.6: Plot of GSM signals prior to channelization showing a broadcast channel in the center and a traffic channel next to it.

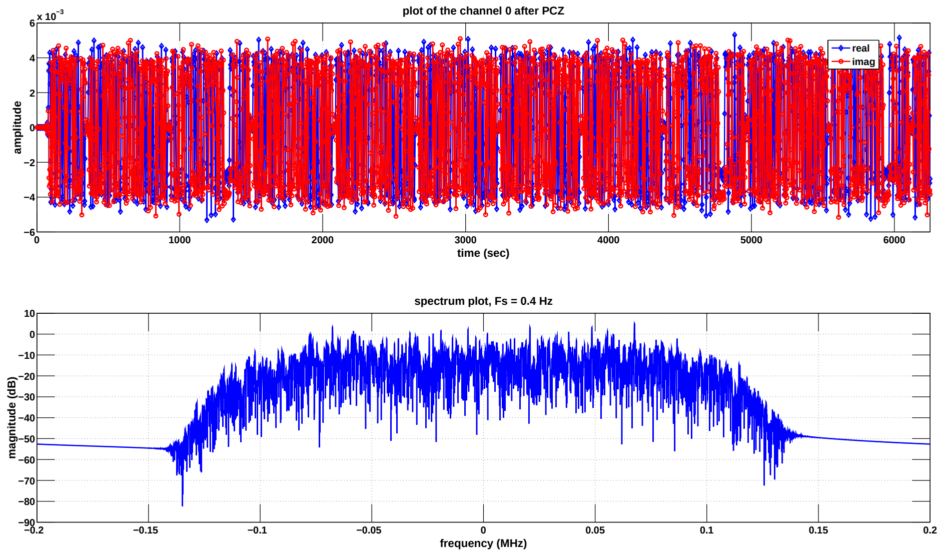

Figure 3.7: Plot of GSM broadcast channel after channelization

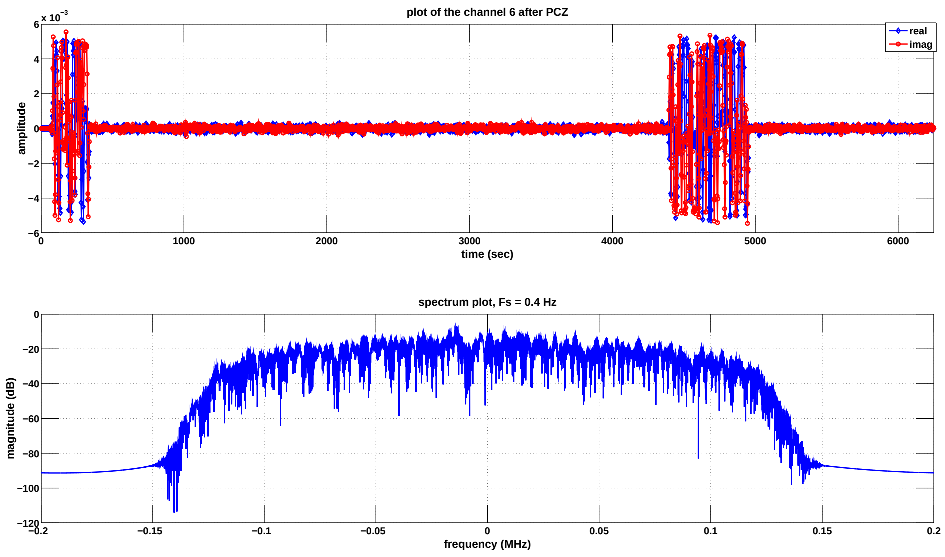

Figure 3.8: Plot of GSM traffic channel after channelization

Table 3.1: Results for one GSM multiframe

	with transfer	without transfer
polyphase channelizer	142.4 ms / 29.2 MSps	118.6 ms / 35.1 MSps
SRC	108.3 ms / 38.4 MSps	92.7 ms / 44.9 MSps
overall	227.8 ms / 18.3 MSps	211.8 ms / 19.6 MSps

Table 3.1. Here, *MSps* stands for mega samples per second.

We modeled and simulated the entire process in MATLAB from the captured over-the-air data. Then we implemented our design in C and CUDA for the CPU and GPU, respectively. We used floating-point precision throughout. We tested our implementation on a single MF, which has 32,500 samples. In Table 3.1, we can see that we achieved high throughput on all of the kernel calls and on the overall processing for the MF. Our polyphase channelizer kernel includes CUFFT, as described in Section 3.4. Since it is highly optimized for the GPU, the execution time for CUFFT is very small and the throughput is over 1 GSps.

In Table 3.1, we show the execution times for each kernel. The targeted latency for our overall runtime measurement was the duration of a traffic MF, which is 120 ms. Our implementation missed this targeted latency only on the polyphase channelizer kernel call with the data transfer time included. Typically, one would leave the data in GPU for further processing so this inclusion of the data transfer time artificially inflates the runtime measurement in the context of practical implementation scenarios. Thus, for practical purposes, our implementation can be viewed as consistently achieving the required latency constraint.

Even though we removed the serpentine shift, we used an FSM to switch between different states in the transfer buffer and this added overhead via nested branch conditions. These branch conditions cause serialization in the GPU, which carries clock cycle penalties. For the arbitrary SRC kernel, we were under the MF duration. For the overall runtime, we were under 240 ms or 2 MF durations. Our main goal is to demonstrate high-throughput using a GPU and our slowest kernel call is over 100 times faster than the data rate while being close to the targeted latency constraint. Overall, we can process all 124 channels and resample all of the channels at the same time in less than 2 MF durations. This demonstrates concretely a major advantage of our GPU-based implementation as a front-end receiver.

Our reference serial CPU implementation exhibited runtime and throughput levels of 1,010 ms and 4.12 MSps, respectively, for one channel. For the all-channel version, it resulted in 1,580 ms and 2.63 MSps. Although the throughput is still high, the long runtime leads to unacceptable latency for real-time application, whereas our GPU implementation can be used in real-time scenarios.

We also performed an experiment that isolated the effect of our approach to buffer indexing as an alternative to using serpentine shift and an input buffer. From this experiment, we measured a factor of 22 (22x) speedup using our indexing scheme compared to the serpentine shift operation. For our GPU implementation, we used the same dimension as we have discussed in Section 3.5. Our implementation resulted in no register spill loads or stores, and SM usage of less than 384 bytes.

Our GPU implementation not only exhibits high throughput, but also latency that is close to the frame time for all channels, not just for a single channel. We note that these measurements are for a desktop GPU, and we primarily focus on the throughout since the latency of the system can be heavily architecture dependent. For example, the latency can be reduced on a mobile GPU or an embedded GPU that is integrated more tightly with the host CPU as compared to a desktop GPU. Also, the memory transfer and access characteristics of an embedded implementation can be significantly different from the desktop version, and can be tailored toward low latency by using techniques such as pipelined or “ping-pong” buffers. Regardless of the selected platform, we were able to demonstrate real-time performance using actual GSM TCH data. In addition, we demonstrated our implementation using floating-point precision in all software components, which is in contrast to the use of fixed-point representations that is typical in hardware solutions. Our use of floating point provides significantly enhanced flexibility and testability, and facilitates rapid prototyping.

3.7 Summary

In this chapter, we presented a GPU-based polyphase channelizer that performs the major tasks of down conversion, filtering, and resampling of a communication channel of interest. In addition, we developed an implementation of sampling rate conversion using a polyphase filter based arbitrary resampler that resamples to any arbitrary rate. We presented a GPU-based front-end receiver architecture that utilizes the parallelism found within these subsystems, and achieves high throughput and low latency. We improved and simplified the channelizer architecture by replacing complex buffer operations with a simple, efficient indexing scheme.

We demonstrated the capability of assigning a channel to a dedicated block of GPU threads, and applying multiple blocks in parallel to simultaneously process large numbers of channels, thereby realizing a novel form of high data rate parallel receiver. Our approach enables design of a system that has the flexibility of a single-channel receiver, and the performance and throughput of an all-channel-in-one receiver. Our solution is a software-based, floating-point, general purpose implementation rather than a fixed-point dedicated hardware accelerator. These capabilities together realize the objective of using a GPU as a flexible and highly parallel device for bringing signal processing capabilities closer to the antenna.

Chapter 4: Implementation of a High Throughput Polyphase Channelizer on GPUs

In this chapter, we propose a novel GPU-based polyphase channelizer architecture that delivers high-throughput and low latency. This architecture has advantages of having reduced complexity, and being optimized for parallel processing of many channels simultaneously, while also being configurable via software. This architecture builds on the design presented in Chapter 3, and improves this design in various ways to simultaneously provide major gains in both throughput and latency. We demonstrate the performance of this new architecture by applying it to channelization of signals based on the 3G wireless standard, UMTS/WCDMA. The work of this chapter was presented in [49].

4.1 Introduction

A modern communication transceiver contains two major components: a radio frequency integrated circuit (RFIC) and a baseband processor. An RFIC is responsible for conversion between analog and digital domain signals, and mixing signals up and down from baseband to some RF. A baseband processor or a radio modem is responsible for handling all of the signal processing tasks and communication protocols. The notion of software radio (SWR) is defined in [1]. This notion of SWR is different from software defined radio (SDR). An SWR is responsible for the entire processing chain between RF and baseband via software, whereas SDR is responsible for the chain between intermediate frequency (IF) and baseband. A modern RFIC can also be fully programmed and reconfigured via software from baseband modem microprocessors.

In SWR systems, signal processing (SP) tasks, such as signal conversion, mixing, resampling, and filtering, are all done in baseband using software in the discrete-time sample domain. An RFIC is then responsible for direct conversion to and from RF. The SWR design process reduces the complexity of the RF front-end, but imposes an increased burden on the baseband processing subsystem. The process enables reconfiguration of communication system features via software, which is much more efficient compared to redesigning banks of dedicated sub-receivers in hardware. This software-intensive design process is also attractive for designers and engineers who can focus more on their particular form of expertise — e.g., RF engineers can focus on RF circuit design, while SP engineers can focus on developing SP algorithms and their associated software implementations.

A majority of the transceiver functionality is contained in the software modem. The goal of the software-based transceiver is to bring the SP functionality closer to the antenna as much as possible, reducing the burden on the RF front-end, and utilizing the full flexibility of software. A software-based modem is particularly attractive over dedicated hardware solutions, such as ASIC- and FPGA-based solutions, due to significantly reduced design time from modeling to implementation to production. One of the major advantages of dedicated hardware is low-power design, but with the advancements made in system-on-chip (SoC) architectures over the years with particular emphasis on low-power design, solutions based on programmable SoC architectures can deliver levels of energy efficiency that are sufficient for many applications (e.g., see [50]). An SoC can be delivered as a complete solution that integrates not only the radio unit but other key units, such as CPU, GPU, and peripheral controller subsystems, as well.

A modern communication system requires multiple users and data streams to be processed simultaneously. A front-end transceiver must be able to transmit and receive multiple channels simultaneously, and a technique known as channelization is used to separate multiple users or channels from a single communication stream. A channelization process is responsible for 3 basic tasks: (1) signal up/down conversion (mixing), (2) sample rate change, and (3) filtering. In this chapter, we refer to channelization in the receiver architecture only, which means that the channelization process is responsible for down conversion, reducing sample rates, and filtering to reject images at the receiver.

A straightforward approach to designing a channelizer is to design a bank of dedicated sub-receivers. Each sub-receiver is allocated to a single channel. Such an approach involves large costs in terms of area, power, and complexity. At the same time, the channels are in general not all used simultaneously, and many of the sub-receivers may be idle at any given time during operation. Such idle sub-receivers are wasteful in terms of power consumption and area.

A more integrated approach is needed to replace such a "room full of receivers" to reduce redundancies, and improve resource utilization. For this purpose, a polyphase channelizer was introduced in [13, 14]. This architecture employed polyphase filter banks (PFBs) and DFT operations to accomplish multiple channelization tasks at the same time. In particular, the PFB was used to perform inner-product (IP) computation for filtering and resampling at the same time, and DFTs were used for mixing signals up or down. We refer to a polyphase channelizer in the receiver chain as a polyphase down channelizer and in the transmitter as a polyphase up channelizer. In this chapter, we focus on polyphase down channelizer implementation, which we will simply refer to it as a polyphase channelizer. The input to a polyphase channelizer is a frequency domain multiplexed (FDM) signal, and the output is a time domain multiplexed (TDM) signal. The input FDM signal can represent, for example, a dedicated channel for one user or a channel that is shared across multiple users using spreading codes within the channel. The output corresponding to each DFT index corresponds to a specific user in the TDM signal.

In this chapter, we demonstrate an important application of GPU technology to SWR systems. In particular, we develop a novel GPU-based polyphase channelizer architecture that delivers high-throughput, and provides reduced complexity and optimized parallel processing of many channels, while being configurable via software. Since baseband modems require SP accelerators that are performing the same SP tasks on incoming streams of data, there is significant data and task parallelism available, which we exploit in our proposed architecture using the intensive parallel processing capability of a GPU. In our proposed design, the GPU can be used as a stand-alone unit or in conjunction with an existing hardware modem. Our goal is to use the GPU as a radio, and bring it as close to the antenna as possible. Such a GPU-based system can reduce the burden on a power hungry baseband modem, and ideally replace the existing modem altogether. Thus, our proposed channelizer architecture simplifies the design, and enhances flexibility, while providing significantly accelerated performances.

The remainder of this chapter is organized as follows. First, we discuss the theory and operation of polyphase channelizer. We then introduce a novel GPU-based approach for high-throughput polyphase channelizer implementation. We develop the optimized mapping of polyphase channelizer functionality onto GPUs for parallel processing of PFB and DFT subsystems. We also introduce a method for assigning a communication channel to a block of threads in a GPU so that simultaneous processing of many channels can be performed in parallel. This method for exploiting parallelism across channels enables implementation of scalable, high-throughput, and real-time parallel transceivers. Finally, we integrate all of the novel methods developed in this chapter and demonstrate their utility using an important wireless communication standard.

4.2 Related Work

We introduced the notion of using the GPU as a radio, particularly as a front-end transceiver, in [41], and in this chapter, we continue to explore the concept of a GPU front-end (GFE) receiver. GPU back-end receivers, which are responsible for channel decoding (e.g., using Turbo and LDPC decoders), are captured in [51, 52]. A modern GPU-powered communication system that uses multiple antenna configurations and a MIMO detector is been presented in [53].

Other related work on application of GPUs to communication system design includes GPU acceleration of FFT computation for channelization [54], integrating GPU technology into a software radio framework [55], accelerating polyphase filters using GPUs [56], and channelization via mobile GPUs [57]. In [54, 57], optimizing FFT and PFB for wideband channelization was introduced using OpenCL. This work investigated implementations that were targeted to different classes of AMD GPUs. It targeted GNU Radio's polyphase filter channelizer and compared the speedups and computation time between CPUs and different GPUs. In this work, we design and optimize our polyphase channelizer on NVIDIA GPUs using CUDA. Our primary goal is to target our implementation toward wireless communication systems, and toward meeting critical performance constraints of such systems — in particular, constraints on throughput and latency.

Additionally, there are various FPGA/VLSI implementations of polyphase channelizers in the literature (e.g., see [47, 58–61]). These works largely focus on optimizing resource usage, such as use of multipliers, and memory.

A preliminary version of this work was presented in [41]. This new chapter goes beyond the developments of the preliminary version by incorporating significant new enhancements to our proposed polyphase channelizer architecture for high-throughput and real-time communication systems. Specifically, we further enforced coalesced loads and stores from GM to SM; spread the work across the GPU more efficiently by enabling increased workloads and scheduling more blocks of threads for the GPU to process; and eliminated some sequential aspects of the underlying algorithms that were present in our preliminary version. Due to these enhancements, the execution time of our new architecture is significantly reduced, well below the target latency. Furthermore, the throughput has been increased significantly, while providing for simultaneous processing of multiple channels.

4.3 GPU-based High-Throughput Polyphase Channelizer

We map our polyphase channelizer algorithm onto a GPU and exploit parallelism found in polyphase channelizer operations. First, we implement a fully parallel PFB on a GPU, with the GPU used here to accelerate IP operations. We then integrate into our GPU implementation a CUFFT kernel. CUFFT, a part of NVIDIA's library of signal processing blocks, is a parallel version of the DFT that is highly optimized for use in CUDA. We process real I-Q values instead of complex values in our GPU implementation.

We demonstrated an approach to high-throughput IP computation using GPUs in [41, 44]. In this approach, we are given an input array from the host CPU that is stored initially in GM. Instead of using an input buffer with dimensions $Q \times M$ to match the PFB, we simply index the necessary input samples. In order to minimize the usage of GM, we first load the data from GM into SM. Each SM contains M groups of samples so that the IP for multiple samples can be computed simultaneously. Since we can access Q samples at the same time, we no longer need a commutator. However, one must be careful to access the input samples in a bottom-up manner to ensure that processing is carried out in the correct order.

The algorithm presented in [41] uses a sequential for-loop to index through the input data. This is a simple approach to access Q samples at a time, but this serialization inside the GPU causes increased latency. If there is a large number of input samples to process, then this loop will dominate over the fast parallel processing inside the loop. We propose a new algorithm that eliminates this for-loop based processing. Specifically, instead of using a for-loop to step through the input, we unroll the loop completely and map each sample to a separate thread. This enforces the notion of SIMT processing in the GPU since we are performing the same operation over and over across the input data stream — i.e., polyphase filtering over the input data. This design optimization eliminates sequential operation. Furthermore, because only one output sample is generated by each thread, the optimization enables the spawning of larger numbers of blocks and threads across the GPU, which helps to improve overall device utilization. Figure 4.1 shows how the data is split and loaded onto the targeted GPU.

Our objectives in further optimizing the design beyond the developments in [41] include exploiting the SM as much as possible rather than reading and writing excessively from and to GM. The optimized kernel design provides a larger workload that is spread across the threads, with each thread encapsulating a lightweight operation. Since the volume of input data exceeds the filter dimension, GPU utilization is increased. Each block or SM is loaded with $TPB + M - 1$ input samples, and each thread is now responsible for filtering M samples. Therefore, in the optimized design, a thread encompasses a multiply-and-accumulate (MAC) operation, which

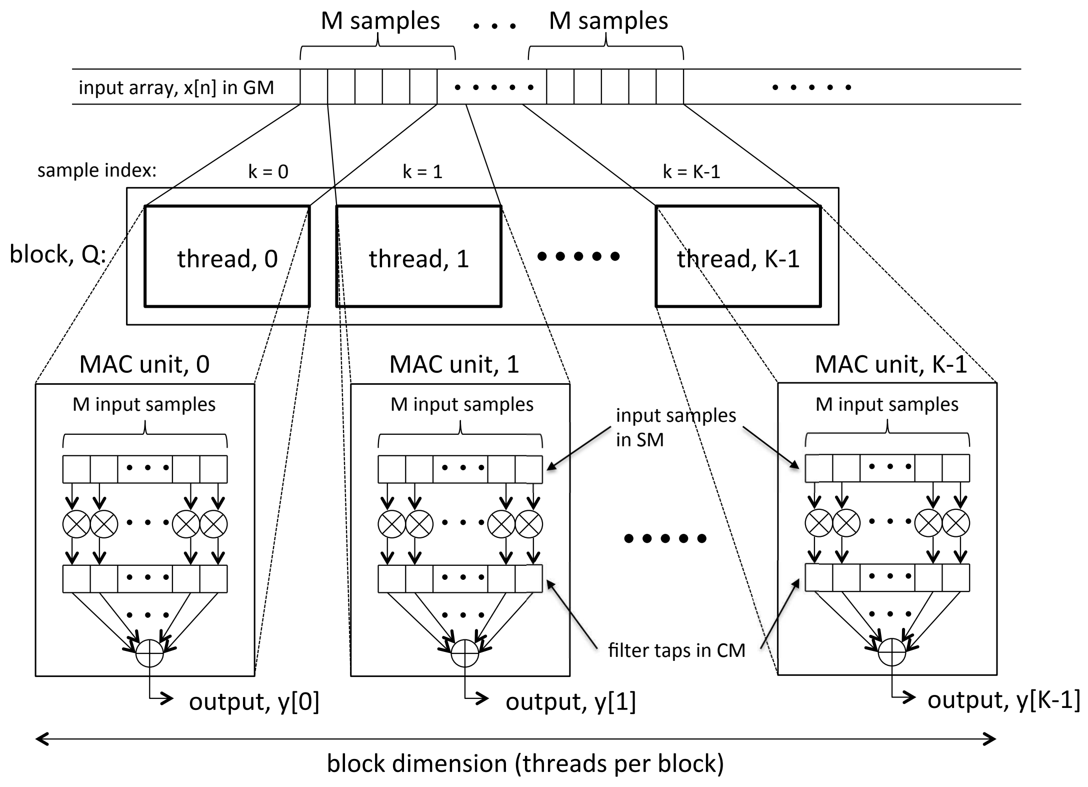

Figure 4.1: An example of how data is split and loaded onto the GPU.

also takes advantage of GPU's fused multiply-add (FMA) operation. This is compared to our previous design, where each thread performed multiplication only. The results of these multiplications were then summed separately using a single thread, which paused the other threads, leading to less efficient GPU utilization and loading of the GPU.

For polyphase filtering, we load the values column-wise, but we operate row-wise. Thus, when loading the data from GM to SM, the data is not coalesced properly, and an additional step is necessary to enforce further coalescing in the IP operation. A kernel is applied to shuffle the data to pre-position the data prior to polyphase filtering. A pseudocode specification of this data shuffling process is

```
$idx = blockIdx.x \times blockDim.x + threadIdx.x$
if $idx < INPUT\_LENGTH$ then
    $out[col + row \times SAMPLES\_PER\_ROW] = in[idx]$
end if
```

Figure 4.2: Pseudocode for data shuffling

shown in Figure 4.2. It is important to note that we read the data in a linear fashion initially, to enforce caching on the read operation, then we write back to GM in polyphase decomposed fashion. This reduces the latency slightly compared to reading the data in polyphase decomposition manner first, and then writing it back linearly. Following this operation, the polyphase filter kernel is called. This kernel now reads the data linearly in coalesced manner to SM. Since we instantiate threads that are multiples of a warp, the access pattern is byte-aligned and linearly read, which further enhances the efficiency of the GPU implementation.

To demonstrate the overall operation of our proposed fully parallel polyphase channelizer design, we provide the pseudocode specification shown in Figure 4.4. Even with an extra kernel for data shuffling, the entire operation is now simplified and further streamlined by eliminating a dominant sequential loop from our previous design. We reshuffle the data back to its original format prior to applying CUFFT for the DFT. After application of CUFFT, the channelization process is complete. Each of the CUFFT output index corresponds to a TDM output stream. All of the the channel outputs are produced simultaneously due to the parallel structure of our proposed architecture. Figure 4.3 shows the overall block diagram of our new,

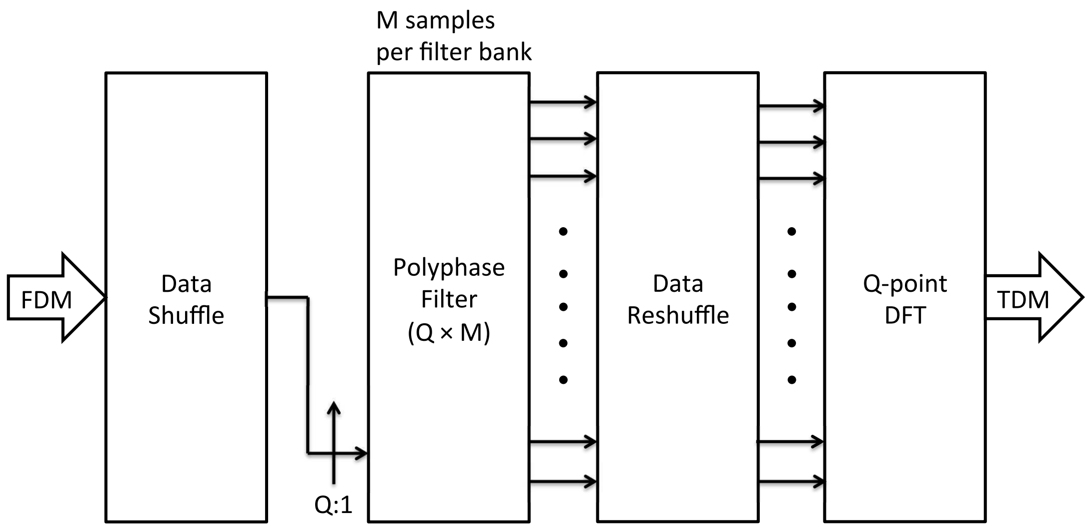

Figure 4.3: Block diagram of GPU optimized polyphase channelizer

optimized polyphase channelizer implementation.

In summary, in this section we have built on our recent developments on polyphase channelizer implementation [41], and incorporated additional design optimizations to further improve performance. The new design optimizations discussed here include minimizing the rate of data transfers, enhancing coalesced access of GM, optimized utilization of SM, and enhanced GPU utilization by reducing thread granularity (operation complexity). The result is a simpler architecture with reduced bottlenecks and elimination of a dominant sequential loop. Collectively, these optimizations result in significant further improvement in throughput and latency.

In the remainder of this chapter, we demonstrate and experiment with our proposed design methods using a wireless communication standard. The results provide concrete insight into the the performance of our GPU-based, multi-channel, parallel transceiver in the context of a practical wireless communication system.

```
$ix = threadIdx.x$
$iy = blockIdx.y$
$pdx = blockIdx.x \times blockDim.x + threadIdx.x$
$idx = iy \times SAMPLES\_PER\_ROW + pdx$
$odx = pdx \times Q + iy$
if $pdx < SAMPLES\_PER\_ROW$ then
    $SM\_REG[ix + M - 1] = in[idx]$
    if $ix < M - 1$ then
        $SM\_REG[ix] = in[idx - M + 1]$
    end if
    $SYNC\_THREADS$
    for $ii = 0$ to $M - 1$ do
        $SM\_MAC[ix] += CM\_COEF[(M-1-ii) \times Q + iy] \times SM\_REG[ix + M - 1 - ii]$
    end for
    $SYNC\_THREADS$
    $out[odx] = SM\_MAC[ix]$
end if
```

Figure 4.4: Pseudocode for polyphase channelizer

4.4 Implementation and Experimental Setup

To experiment with our proposed new polyphase channelizer design, we target an important 3G wireless standard, the UMTS air interface, WCDMA [62]. The radio frame duration of WCDMA is 10 ms, which is further divided into 15 time slots per frame. In our experiments, we consider the front-end of a receiver at the base station or at the associated user equipment to evaluate our optimized polyphase channelizer design.

WCDMA/UMTS has a set of allocated frequency bands or operating band numbers. Each operating band has a center frequency and a bandwidth (BW) associated with it. Each band can occupy several tens of MHz, as much as 80 MHz. WCDMA is a spread spectrum system that has a data (chip) rate of 3.84 MHz and occupies approximately 5 MHz of BW. One of the common BW levels of UMTS is 60 MHz. Given that a modern RFIC can handle an instantaneous BW of more than 60 MHz, we assume that at the input to our GFE, a 60 MHz wide BW is presented. Within this wide BW, there can exist multiple WCDMA signals with 5 MHz channel spacing. Therefore, we have at most 12 WCDMA channels present, as shown in Figure 4.5. We process all of the available WCDMA channels (up to 12) simultaneously using our polyphase channelizer implementation.

Our approach to using polyphase channelizer here works well given a wide input BW, equal channel spacing, downconversion, and the ability to reduce the sampling rate at the output of the polyphase channelizer simultaneously using a prototype filter and DFT. In addition, this particular type of channelizer converts FDM channels into TDM channels. The GPU-based polyphase channelizer provides a highly parallelized and efficient channelization option, which we demonstrate in this chapter for a realistic system scenario. More details on this demonstration are discussed in the following section, which covers experimental results and analysis.

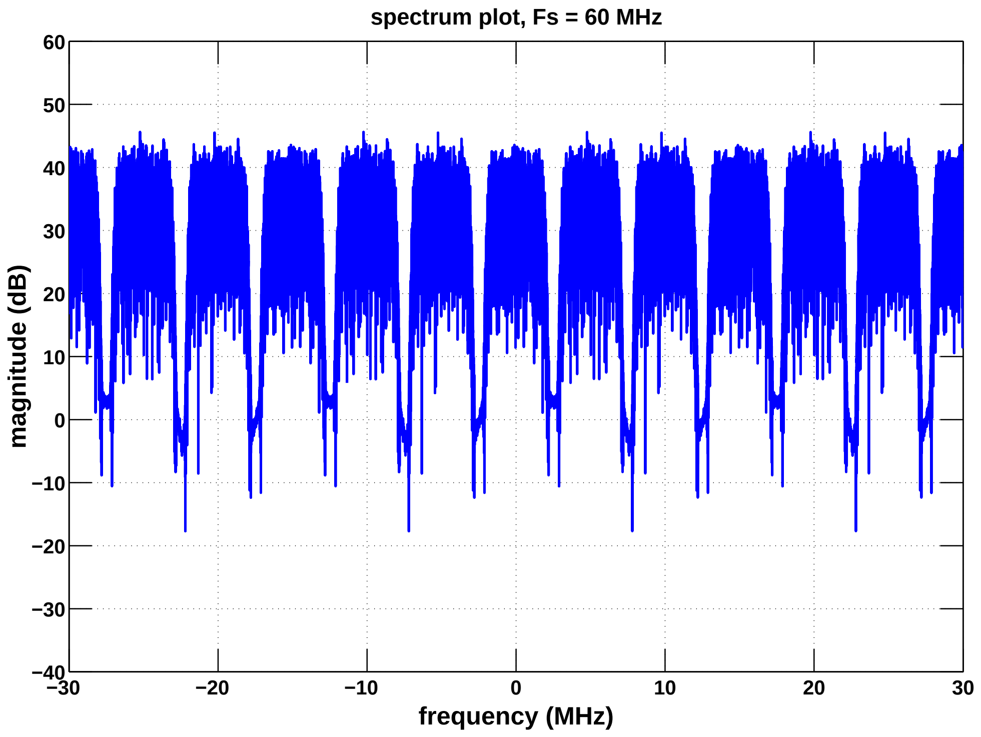

Figure 4.5: 12 WCDMA channels are present in 60 MHz wide bandwidth.

We design our prototype filter using an equiripple FIR filter with order, $N = 192$. We design the filter such that the passband covers the WCDMA band up to 3.84 MHz, and the stopband is near 5 MHz. The passband ripples and stopband attenuation are 0.05 dB and 70 dB respectively. This gives us non-overlapping polyphase filters, unlike the overlapping polyphase channelizer filter design that we presented in [41] for GSM. Since WCDMA uses QPSK modulation, it is important that we preserve the passband and that we do not overlap in the filter transition region, unlike GSM's GMSK modulation in [41]. A sample plot of our filter design is shown in Figure 4.6. Across our 60 MHz system BW, there is a total of 12 polyphase filters and channels side-by-side. This can be viewed as an example

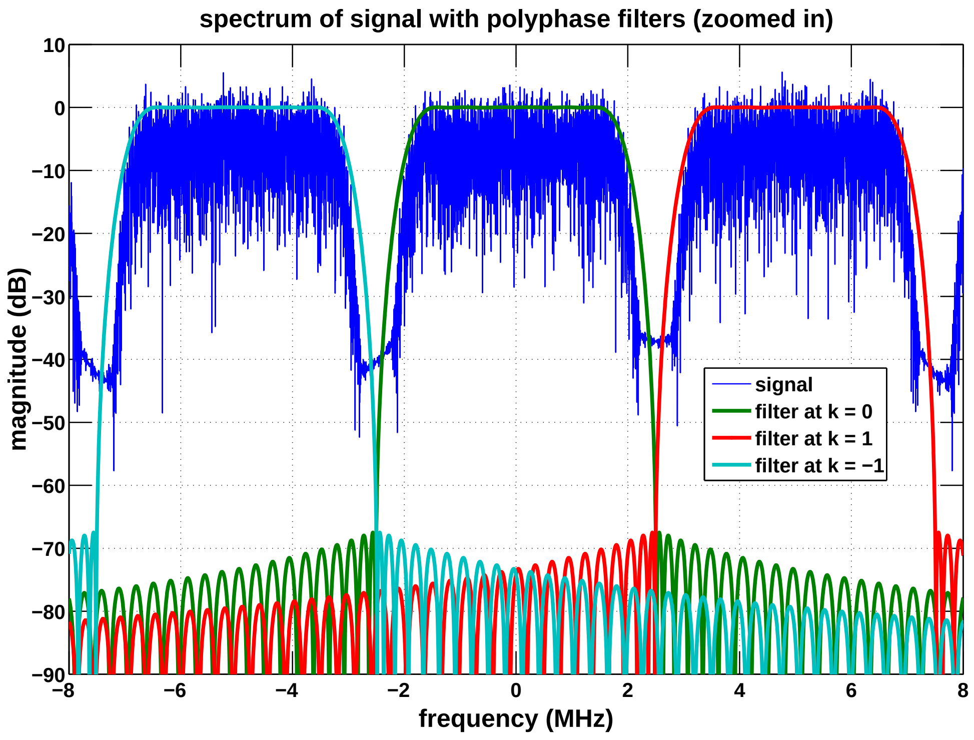

Figure 4.6: A zoomed-in plot of 3 WCDMA channels with non-overlapping polyphase channelizer filters spaced apart at 5 MHz.

where a maximum number of channels is present in a band to maximize the network capacity.

Since there are 12 channels to process, we decompose our polyphase channelizer into 12 rows or PFBs, resulting in 16 subfilter coefficients per row for our prototype filter length of 192. Therefore, we have a $Q \times M$ IP matrix, where $Q = 12$ and $M = 16$. The decimation rate or number of rows, Q is also the number of DFT points. Here, $M = 16$, which is a half of the GPU warp size. After application of the polyphase channelizer, the output sampling rate should be $60/Q$ or 5 MHz, which matches the desired channel spacing of WCDMA. Since we have Q input samples being presented and Q IP values for a Q-point DFT, which produces Q channel outputs, we have a maximally decimated PFB channelizer. In addition, we process each channel independently, thereby realizing a fully parallel transceiver.

Now that we have all of the parameters in place, we map our WCDMA-targeted polyphase channelizer using the algorithm presented in the previous section. First, we prepare the input data for coalesced access with polyphase filtering. Given 60 MHz of BW and a 10 ms radio frame duration, we pre-process 600,000 samples into the desired 2D matrix for polyphase filter operation in the polyphase channelizer. We shuffle or reshape the 1D input array into a 2D matrix, as discussed earlier. This can be viewed as a row-major order that is transposed or simply a matrix that is loaded column first, but operated on across the rows, as given by the polyphase decomposition. This operation takes advantage of cached read accesses each time, and block processing of input data. In addition, it enhances coalesced reads from GM to SM in the polyphase filtering process. This is implemented as a separate CUDA kernel prior to a filter kernel.

Earlier, we described our previous algorithm as being for-loop-dominant and indexing through the input array sequentially. Performance can be improved significantly when it is possible to parallelize this sequential process. For this purpose, we unrolled the loop completely and divided up the workload across more threads and blocks to utilize large numbers of blocks and cores in the targeted GPU.

While our original method had worked in the context of standards such as GSM that have longer radio frame durations of 120 ms, such a method is not well-suited to 3GPP radio frames, which have durations that are 12 times shorter at

10 ms. Operation within 3GPP poses further challenges since faster processing is required. The new design presented in this chapter maps the polyphase channelizer operation more efficiently for practical operation within the context of 3GPP. The workload is spread across the GPU more evenly such that the GPU has a large number of lightweight threads operating, which helps to improve GPU utilization, as discussed earlier.

Next, we discuss the integration of the different components discussed above to construct our overall polyphase channelizer design. We exploit SIMT in our kernel dimension designs. We can reshape our kernel dimension each time, dividing up the workload evenly across different kernels. Multiple kernel calls are used to split the data for different purposes. First, our reshape or data shuffling kernel divides up one frame worth of data, composed of 600,000 input samples, by 512 TPB, yielding 1,172 blocks. This is a 1D split that assigns one thread per output sample. For the PFB kernel, this same data set is divided into 12 channels, with 50,000 samples per channel. Each channel is then further divided by 512 TPB, which yields 98 blocks. Since there are 12 channels, we use a 2D data split, where the x-direction corresponds individual samples within a channel, and the y-direction corresponds to specific channels. Thus, the total lengths of the x- and y-directions are N_x and N_y, respectively, where N_x is the number of samples per channel, and N_y is the number of channels.

This decomposition maps 600,000 input samples into one sample per thread across the 2D grid dimension. Here, each thread is responsible for 1 sub-filter operation, therefore, it will compute $M = 16$ multiply-and-accumulate (MAC) operations per thread. Each input sample is read into SM along with $(M-1)$ previous sample points to perform filter operation. The independent MAC operations are performed by each thread (i.e., 512 threads perform 512 MAC operations with each thread accessing 16 samples). This further improves the performance by increasing the utilization of SM instead of using registers or LM. The filter coefficients are stored in CM so that they are cached and broadcast throughout the entire kernel for fast, read-only operation. The output of the IP is then reshuffled when written back to GM for the subsequent CUFFT operation. The last kernel call in the sequence is the CUFFT, which outputs the TDM data for each of 12 individual channels at the same time.

Thus, to handle the more demanding processing required when applying our GPU-based polyphase channelizer design in 3GPP, we see that our new design employs much larger numbers of blocks and threads per block. In particular, under these design constraints, our previous design instantiates 12 blocks and 16 threads per block, which are sufficient for the GSM context in which the design was originally applied, but leaves large numbers of unused blocks and threads in the GPU. This low utilization of GPU resources is problematic under the more severe performance constraints of the 3GPP communication system targeted in this chapter. Additionally, using multiple kernel calls provide significant flexibility to reshape the kernel dimension each time, which is a form of flexibility not found in dedicated hardware solutions.

Table 4.1: Experimental results (run-time / throughput)

	with transfer	without transfer
shuffle	2.085 ms / 575.5 MSps	0.116 ms / 10,357.5 MSps
channelizer	2.868 ms / 418.5 MSps	0.734 ms / 1,633.6 MSps
overall	2.901 ms / 413.5 MSps	0.856 ms / 1,402.0 MSps

4.5 Results and Analysis

For our experiments, we employed NVIDIA's GeForce GTX 680 as the target GPU to implement our design. This GPU is based on the Kepler architecture, which has 1,536 CUDA cores (8 SMXs with 192 CUDA cores per SMX), 2 GB of GDDR5 memory, 64 kB of CM, and 48 kB of SM. It has 256-bit memory bus width, 192 GB/s bandwidth, and over 3,000 GFLOPS speed. We used the latest CUDA driver version 6.0, and a compute capability of 3.0. We summarize our results in Table 4.1 for both throughput and run-time. We also present our results with and without memory transfer between the CPU and GPU. Our target latency for real-time operation is 3GPP's radio frame length of 10 ms, and our calculated speedups are based on a sampling rate of 5 MSps (mega samples per second).

Our emphasis here is on high throughput and low latency polyphase channelizer implementation. We use 32-bit floating-point precision throughout the experiments. We test our implementation on a collected WCDMA band of 60 MHz for 10 ms, where each WCDMA channel occupies 5 MHz of BW. Thus, the overall 60 MHz bandwidth is channelized into 12 channels. We achieve a high throughput on all of the kernel calls. The polyphase channelizer kernel call includes CUFFT since CUFFT is highly optimized and available as a part of the CUDA software development kit (SDK). As one can see from the results, when memory transfer is involved, it dominates the overall run-time. Therefore, unnecessary transfer of data between the CPU and GPU is highly undesirable.

Our implementation results in no register spills, SM usage of 8,312 bytes, and CM usage of 360 bytes. As one can see from Table 4.1, all of our kernel calls are executed with performance that falls within our target latency of 10 ms, even with the overhead of data transfers taken into account. Without data transfers, the kernels ran under 1 ms. The overall kernel call (which is the slowest of all) achieves over 280x speedup compared to the 5 MSps sampling rate. Since this is a front-end and first stage of baseband processing, one would typically leave the data in the GPU for further processing, and only transfer data back to the CPU when needed after such further processing is complete.

We note that the data shuffle kernel achieves an overall occupancy of 84.5%; a global memory load efficiency of 100%, which is as expected due to our use of linear coalesced reads; and a global memory store efficiency of 33%, which is due to shuffling of data for polyphase decomposition, as we discussed earlier. Nearly all of the kernel execution time is spent on load and store operations since there are no arithmetic operations involved.

The polyphase channelizer kernel, which encapsulates the PFB operation, achieves an overall occupancy of 95.6%, and GM load efficiency of 99.8%. Due to our use of the data shuffle kernel, we are able to enforce coalescing for nearly 100% of the global read operations. Without our use of the data shuffle kernel here, we expect that the GM load efficiency would be much lower due to irregular read patterns for polyphase filtering. For the polyphase channelizer kernel, SM efficiency is 99.8%, but GM store efficiency is nearly 0%. This low level of GM store efficiency is expected; it is due to non-coalesced writes from re-shuffling data after MAC operations across different blocks and threads.

The PFB operation utilizes full fused multiply-add (FMA) floating point operations in the kernel. The GPU excels in such computations [63]. Because our kernels under-utilize available computational resources to some extent, the kernels are memory bounded (at the L1 cache). However, this is not a major bottleneck in our implementation since our main goal is to process data channels in real-time, and the implementation meets these objectives in terms of throughput and latency. We note here that our experiments apply to a single instance of a data set rather than a continuous stream of data. Applying a continuous stream of data could lead to higher levels of utilization for the available computational resources. A useful direction for future work is the further exploration of the potential of our implementation in the context of continuously streaming data.

We compare the performance of our previous polyphase channelizer design [41] with the new design that we propose in this chapter. We emphasize that although our previous design (from [41]) exhibits lower performance compared to our new design, the previous design successfully met the performance constraints of GSM,

Table 4.2: Comparison of polyphase channelizer designs (run-time / throughput)

	with transfer	without transfer
old channelizer	52.591 ms / 22.8 MSps	50.808 ms / 23.6 MSps
new channelizer	2.868 ms / 418.5 MSps	0.734 ms / 1,633.6 MSps
speedup	18.3x	69.2x

which is the primary standard to which it was targeted. The new design introduced in this chapter has been developed by building on the experience and insights gained from the previous design and targeting the more stringent constraints of 3GPP communication.

Table 4.2 demonstrates that the previous design cannot achieve the 3GPP real-time latency constraint of 10 ms, and in fact, its performance is at least 5 times slower than what is required for the communication system performance targeted in this work. In contrast, our new method achieves the real-time latency constraint, even with memory transfer, as discussed above. Furthermore, without memory transfer, the new design runs under 1 ms, which is near the slot time of 0.667 ms. In other words, the time to process a complete frame is less than 2 times the time required by a single slot in the frame. Overall, our new polyphase channelizer design achieves significantly improved throughput, and a speedup of 326x compared to the sampling rate of 5 MSps. The new design is also nearly 70x faster than the previous design, as shown in Table 4.2.

As expected, the data shuffling kernel exhibits high performance, since it is a simple data swap; however, the new design allows us to combine this data swapping functionality with the benefits of caching and coalescing. Overall, the new polyphase channelizer design provides improvement over the previous one by eliminating serial processing of the for-loop and providing more thorough enforcement of memory coalescing. Additionally, the workload in the new design is spread much more evenly throughout the GPU, and each thread encompasses a fine-grained operation (MAC) utilizing GPU's FMA floating-point operations.

A limitation of our proposed new design is the output data rate, which must be increased with some amount of resampling to achieve the WCDMA data rate. Given the 5 MHz sampling rate at the output of the polyphase channelizer, we would need to resample to at least twice the sample rate of the WCDMA data rate, which is 3.84 MSps. Therefore, a resampler is needed to dynamically achieve such a fractional rate. We presented a GPU-based arbitrary resampling method using polyphase filters in [41]. However, due to serialization within the underlying resampling approach, this approach does not allow us to achieve the target latency when it is integrated with our new polyphase channelizer design. Integrating a suitable resampling subsystem at the output of our proposed new polyphase channelizer design is a useful direction for further investigation.

4.6 Summary

In this chapter, we presented a novel GPU-based polyphase channelizer that achieves high-throughput and low latency. The new architecture eliminates sequential processing, and spreads the processing workload evenly across large numbers of blocks and threads in the targeted GPU. Furthermore, the design incorporates preprocessing of the input data to thoroughly enforce caching and coalescing prior to polyphase filter operation. We demonstrated our application of GPU technology as the basis for front-end transceiver implementation by processing multiple channels simultaneously, and exploiting data parallelism across different channels, which provides large increases in throughput.

Our proposed new GPU-based polyphase channelizer design provides the high performance of a dedicated single receiver using a fully-integrated receiver structure, and without sacrificing flexibility. We demonstrated our design on an important wireless communication standard and demonstrated large speedups in both throughput and latency. We also compared the performance to that of a previous polyphase channelizer design, and demonstrated nearly 70x improvement, while providing detailed analysis of how such speedup improvements have been obtained.

Collectively, the advances presented in this chapter make use of off-the-shelf GPU devices as floating-point software radios that can compete in many design scenarios with fixed-point dedicated hardware radios, and help to bring GPUs one step closer to the antenna.

Chapter 5: Implementation of a Multi-channel Arbitrary Resampler on GPUs

In this chapter, we continue to build on our notion of high performance GFE. We propose a new multi-channel arbitrary resampling approach that is designed for GPU implementation and delivers high throughput, low latency, and high accuracy. This architecture uses the distinctive architectural features of GPUs to compute arbitrary resampling points on-demand without explicitly calculating the resampling index. Such explicit calculation can be computationally expensive and its elimination is a useful feature of our resampling approach.

In the development of our new resampling architecture, we extend traditional implementation techniques for one-dimensional signal processing to multiple-dimensions by processing multiple channels across different frequencies simultaneously. This yields an implementation structure that is well-suited to optimized mapping on GPUs. We demonstrate the performance of our resampling architecture by targeting the 3.5G wireless standard UMTS/HSPA, and resampling multiple UMTS signals simultaneously. Part of the work in this chapter has been presented in [64]. The developments of [64] are included and extended in this chapter.

5.1 Introduction

In wireless communication systems, a spectrum band consists of multiple channels and signals that employ different standards and specifications. A transceiver (TRX) is designed to transmit and receive a channel-of-interest (COI) but in the presence of other channels and users. Therefore, multiple TRXs are needed to service different channels and requirements. Ideally, a single TRX should service all of the COIs, however, it is difficult to implement such a single TRX due to cost, complexity, and different system requirements such as sampling rates, different bandwidths, modulations types, access schemes, etc. Regardless, a front-end TRX must be able to process multiple channels simultaneously. A single, unified approach is preferable to replace such a large number of sub-systems. A channelizer can be used to separate multiple users or channels of a wide-band spectrum. A simple yet efficient channelizer is needed to separate a COI from the rest of the channels. Following a channelization, a resampling is necessary to accommodate different data rates, and ideally resample to 2-4 times the data rate for further processing, such as synchronization, equalization, and channel decoding.

In this chapter, we introduce a flexible front-end TRX, particularly as a receiver (RX) that is highly parallelized for processing multiple COIs at the same time. We simultaneously channelize multiple channels in a band, followed by an arbitrary resampling of all the channels. We employ a fully-software-based solution using GPUs without needing to design dedicated hardware or application specific processors. The resulting architecture is a highly efficient, multi-channel GPU front-end (GFE) transceiver that delivers high throughput while achieving low run-time latency.

The remainder of this chapter is organized as follows. First, we review the theory and operation of channelizers, and arbitrary resampling. We then introduce our novel GFE system that simultaneously channelizes and resamples all of the input channels at the same time. We demonstrate an implementation of our proposed new architecture by targeting a relevant wireless communication standard and profile our system under the associated latency constraint while massively increasing the throughput.

5.2 Related Work

The notion of using the GPU as a radio, particularly as a GFE transceiver, was introduced in [41], which presented channelization and resampling of multiple channels. However, the speedups from this approach came from highly optimized and parallelized filter operations that contained serial loops. This form of parallel implementation was effective for processing GSM radio frames. However, the serialization in the loops presented a bottleneck that precluded application to standards with significantly higher data rates and lower latencies.

In [49], a high-performance channelizer was introduced by unrolling loops completely and spreading the computational load throughout the GPU more evenly compared to the design presented in [41]. This improved design resulted in a low-latency channelizer that processed multiple UMTS/WCDMA channels at the same time. Other related work on GPU front-end design includes GPU acceleration of FFT computation for channelization [54, 57], GPU acceleration of PFBs [56], and GPU acceleration of a polyphase interpolator for timing recovery [44]. Prior research on GPU back-end receiver implementation includes designs for a channel decoder used in HSPA+ and LTE [65], and for MIMO detection [66].

A GPU-based arbitrary sampling rate conversion (ASRC) was presented using a combined polyphase interpolator and polynomial approximation using linear interpolation in [41]. A GPU-based arbitrary resampling using DFT and texture memory was introduced in [64]. In this chapter, we extend the work of [64] by investigating an alternate GPU-based approach to resampling using time-domain interpolation and arbitrary resampling. In [64], a DFT-based interpolation was used to provide frequency-domain interpolation. Although this provides a filter-free design and simple approach for integer interpolation, its performance is affected by prime number factors when performing FFT operations. Conversely, a time-domain interpolation provides significantly improved performance and flexibility through custom filters for interpolation and through the removal of restrictions on the interpolation ratios that can be supported. In addition, we process the full radio frame, including multiple slots, and multiple users, compared to the real-time resampling of a single slot for a single user presented in [64]. Furthermore, this chapter introduces a novel method to provide optimized processing for multidimensional, multirate filtering. This optimized filtering approach performs aggregation of channels to provide much higher BW, which in turn provides higher throughput while keeping the overall latency low.

Much of the related work for channelizer design is targeted to dedicated hardware or FPGA implementation (e.g., see [14, 47, 59]). Similarly, for ASRC, FPGA implementation and optimization for trade-offs between speed and memory requirements are discussed in [22, 25, 26, 67–69].

In hardware implementation targeted to wireless communication systems, performance criteria such as throughput and latency may be met by increasing clock speed, and assigning more resources, such as memory and multipliers. In this chapter, we aim to provide a high performance front-end TRX that meets strict real-time constraints in a purely software-based implementation on commercially available GPU devices. We realize these objectives through efficient algorithm mapping onto the targeted GPU architecture by exploiting parallelism to spread the TRX processing workload more evenly across the GPU. The resulting GPU-based TRX architecture is a flexible software implementation, and an adaptive system that can channelize and arbitrarily resample all input channels simultaneously with massive throughput increase and reduced latency for real-time communication.

5.3 GPU-based Multi-channel Arbitrary Resampling

5.3.1 Channelizer Subsystem

A high-throughput GPU-based polyphase filterbank (PFB) channelizer was presented in [49]. In this chapter, we adapt this channelizer as a part of our new multi-channel arbitrary resampling GFE design. The adapted channelizer design provides an efficient subsystem for channelizing multiple channels. We first implement the PFB by decomposing a 1D FIR filter into a 2D structure of dimensions $Q \times L$, where Q is the number of channels or PFB rows and L is the sub-filter length for each row. An inner product (IP) matrix is formed as shown in [49], and the data is split across SM evenly to promote SIMT processing. Each thread is now responsible for one multiply-and-accumulate (MAC) operation. This architecture maps each output sample to a thread, promoting a lightweight operation across the GPU grid which increases occupancy. A high order of parallelism is achieved at the thread level and across each row of blocks, where each channel is processed independently. We make improvements to the algorithm shown in [49] by eliminating SM-based MAC operations and replacing them with faster register-based operations. Upon completion of this IP operation, the result is written back to GM in a more coalesced manner by using complex floating point values instead of separate, real I-Q values shown in [49]. We provide a pseudocode description of this optimized filtering method in Figure 5.1.

Following this efficient channelization, an arbitrary resampling is necessary to dynamically adapt to different data rates of the desired system specification. Although a PFB channelizer is capable of performing arbitrary resampling, it is a limited operation that is efficient primarily when the resampling rate is a power of 2. Our objective is to avoid being constrained by such a ratio, and instead to be able to resample at any rational ratio, including fractional ratios. Our approach to ASRC in this chapter is to perform an integer interpolation using a polyphase interpolator, and then use the GPU's texture memory (TM) unit to automatically compute the fractional resampling points on all the channels.

```
$tix = threadIdx.x$
$biy = blockIdx.y$
$rdx =$ row index
$idx =$ global (row) input index
$odx =$ global (row) output index
**if** $rdx < SAMPLES\_PER\_ROW$ **then**
  $SM\_REG[tix + L - 1] = in[idx]$
  **if** $tix < L - 1$ **then**
    $SM\_REG[tix] = in[idx - M + 1]$
  **end if**
  $SYNC\_THREADS$
  **for** $ii = 0$ **to** $L - 1$ **do**
    $cplx\_sum \mathrel{+}= CM\_COEF[(L-1-ii) \times Q + biy] \times SM\_REG[tix + L - 1 - ii]$
  **end for**
  $out[odx] = cplx\_sum$
**end if**
```

Figure 5.1: Pseudocode for improved filter operation in polyphase channelizer.

The first stage of integer interpolation is necessary to provide adequate intermediate points for the second stage of interpolation [64]. The second stage in turn employs a polynomial filter to compute the fractional resampling points dynamically. Without some sort of an integer interpolation first, the polynomial approximation is inadequate. On the other hand, without polynomial approximation, we would have to interpolate at a very high rate, which is computationally wasteful since most of the samples would be discarded. Our objective is to interpolate enough initially, and then dynamically determine resampling points-of-interest, thereby reducing computational waste.

Instead of providing resampling on a single channel and one subset (i.e., a slot) of the radio frame, as shown in [64], we extend this method to process the full radio frame. Furthermore, the design introduced in this chapter further resamples all channels simultaneously for all user outputs of the channelizer. This results in a multi-dimensional and multi-channel arbitrary resampler. Additionally, since the channels are independent, we can parallelize the PFB rows, which distributes the workload across the GPU, which increases occupancy and provides a large increase in throughput.

5.3.2 Integer Interpolation Subsystem

In this section, we present details on an optimized GPU-based polyphase interpolator. First, we decompose a 1D FIR filter into a 2D PFB with dimensions $P \times M$, where P is the interpolation rate and M is the sub-filter length for each interpolant. Therefore, if Q denotes the number of channels, then there are $(Q \times P)$ outputs for all of the channels after interpolation.

In order to efficiently perform interpolation on the GPU, we load M input samples into an SM buffer. These samples are broadcast within a block since the same input buffer is used to generate P interpolants for each filter operation. Here, we use SM as a user-enabled cache to enhance multiple read operations. We fetch $(M-1)$ previous samples for each thread such that each thread computes a MAC operation across M samples. Using this approach, each thread exploits the GPU's fused multiply-add (FMA) operation. Similar to our improvements in channelizer MAC operation described above, we only use registers to accumulate the IP results. This way, each thread is responsible for only one accumulated output.

Instead of using a commutator to provide input samples one at a time, we employ vector processing to process the entire block of input samples and simultaneously produce all of the corresponding interpolated outputs. This way, no serial input indexing nor serial-to-parallel conversion is required, and the entire operation is fully parallelized for higher throughput. Once the IP operation is complete, the data is written back to the GM using complex data format for better interleaving of data and coalescing of write operations.

We provide additional decomposition to process multiple channels using the multi-dimensional parallel processing capabilities available on the GPU. A GPU kernel is broken up into a grid of blocks and a block is made up of a group of threads. First, we employ a 2D decomposition of a channel and (data) sample index such that channels are mapped to the y-direction of blocks in a grid, and the rest of the input samples are mapped to the x-direction of blocks. This way, the entire output of the channelizer is evenly mapped to the 2D grid of blocks.

With the grid of blocks determined in this way, an interpolation by a factor of P is performed for each row and on each sample. To achieve this interpolation efficiently, we apply additional decomposition within a block of threads. Because we produce P times more samples than the input samples, we decompose our block dimension such that in the y-direction, it contains the interpolants, whereas in the x-direction it accesses the input samples of the block. This way, we contain all of the interpolants generated by the associated channel in one block. This in turn allows us to simplify indexing of samples; keeps the blocks in the y-direction independent from other rows or channels for completely isolated processing; and enables caching of any shared data within the block using SM.

The resulting operation is a multi-dimensional, multi-channel polyphase interpolation operation. All of the channels are now interpolated at the rate of P, and are ready to be resampled at any arbitrary fractional points. Providing these intermediate interpolants further enhances the accuracy of polynomial filtering via TM. In addition, the commutator operations are eliminated, and the entire set of interpolants is calculated at once. In Figure 5.2, we provide a pseudocode of our approach to multi-channel polyphase interpolation.

5.3.3 Arbitrary Resampling Subsystem

Following the polyphase interpolation stage, we perform a time-varying, polynomial approximation to compute the fractional resampling point, Δ. The GPU texture unit is capable of performing nearest neighbourhood (NN) or linear (LN) interpolation directly and automatically, without the need for additional filters, complex control algorithms, nor manual computation of Δ. We adapt this capability in our ASRC architecture instead of designing our own time-varying filters. Using

```
$tix = threadIdx.x$
$tiy = threadIdx.y$
$rdx =$ row index
$idx =$ global (row) input index
$odx =$ global (row) output index
**if** $rdx < SAMPLES\_PER\_ROW$ **then**
  $SM\_REG[tix + M - 1] = in[idx]$
  **if** $tix < M - 1$ **then**
    $SM\_REG[tix] = in[idx - M + 1]$
  **end if**
  $SYNC\_THREADS$
  **for** $jj = 0$ **to** $M - 1$ **do**
    $cplx\_sum += CM\_COEF[jj \times P + tiy] \times SM\_REG[tix + M - 1 - jj]$
  **end for**
  $out[odx] = cplx\_sum$
**end if**
```

Figure 5.2: Pseudocode for multi-channel polyphase interpolator.

the GPU texture unit, we perform arbitrary resampling across the entire input array to the desired output rate. We only assign a single output sample to a given thread, which spawns more threads in the GPU and we can compute the output immediately from the TM's hardware unit.

To handle the multiple channels and multi-dimensional structure in our re-

```
$idx$ = global (row) input index
$idy$ = global (column) input index
$odx$ = global (row) output index
$\Delta = P \times F_s^{in} / F_s^{out}$
**if** $idx < OUTPUT\_LENGTH$ **then**
    $out[odx] = tex2D(texRef, 0.5f + \Delta \times idx, 0.5f + idy)$
**end if**
```

Figure 5.3: Pseudocode for 2D texture memory-based, arbitrary resampling.

sampling subsystem design, we extend the 1D TM-based ASRC approach in [64] by performing 2D filtering in TM. However, instead of performing a 2D bilinear interpolation, we perform 1D linear interpolation — i.e., across samples in the x-direction only, and not across multiple channels in the y-direction. This interpolation approach represents an additional novel aspect of our design, where we exploit the GPU's multi- dimensional processing capability and indexing schemes to selectively process samples of interest. Figure 5.3 provides a pseudocode summary of our design for multi-channel arbitrary resampling using TM. Here, the texture reference, denoted *texRef*, contains the intermediate interpolants. We provide a block diagram of our complete GFE design in Figure 5.4.

5.4 Implementation and Experimental Setup

We demonstrate our proposed GFE implementation by targeting WCDMA and HSPA for UMTS [62,70], which is an important 3/3.5G wireless communication

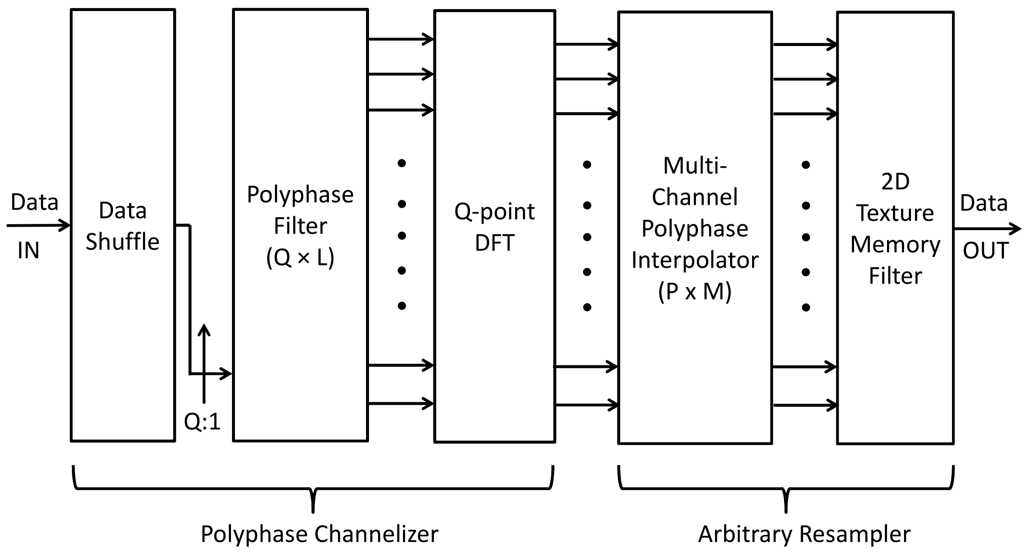

Figure 5.4: Block diagram of complete GFE design including channelizer and multi-channel arbitrary resampler.

standard. The radio frame duration of UMTS is 10 ms, which is further divided into 15 slots per frame, giving a slot duration of 0.667 ms. A shorter frame length of 2 ms is introduced for HSPA. A single UMTS frame occupies 3.84 MHz of bandwidth.

Our objective is to maximize the number of carriers or channels that can be processed in the GFE. The current frequency bands defined by UMTS can occupy as much as 80 MHz of BW. With a typical channel spacing of 5 MHz in UMTS, we can process up to 16 channels in this wide BW. The channels do not overlap in order to avoid interference. Therefore, a PFB channelizer is a suitable choice, since it exhibits equal channel spacing with non-overlapping channels.

We first design a prototype filter to be used in the channelizer. The decomposed polyphase filter has Q channels and sub-filter length $L = 16$. Therefore, the filter length can vary depending on the number of channels it processes. Here, since we are processing 16 channels, the total length of the filter is 256 taps. We employ an equiripple FIR filter with 70 dB attenuation and 0.04 dB passband ripple. The output of the channelizer presents all 16 channels simultaneously. However, due to the equal spacing of the output channels, the output sampling rate is fixed at 5 MHz, while we desire at least 7.68 MHz — i.e. twice the bandwidth of UMTS — for further baseband processing. Therefore, a resampler is a needed to convert the derived 5 MHz rate to the desired 7.68 MHz rate.

Instead of using a rational resampler or a sequential, combined PFB-based ASRC as shown in [22,41], we apply a 2-stage approach where an integer polyphase interpolation serves as an initial interpolation. This interpolation stage enhances the polynomial approximation in the following stage using the GPU's texture memory (TM) filtering option. The prototype filter for the interpolator has 512 taps, which are decomposed into a $P \times M$ polyphase matrix, where $P = 16$ and $M = 32$. We employ an equiripple FIR filter with 110 dB attenuation and 0.002 dB passband ripple. Following this integer interpolation, we bind the interpolants to a CUDA Array and perform either NN or LN interpolation in a TM kernel. This TM kernel automatically fetches and computes fractional resampling points directly, which eliminates the need for manual computation of the resampling points.

Because we are able to process multi-carrier (MC) channels simultaneously, our architecture supports MC-WCDMA and/or MC-HSPA options, and realizes carrier aggregation (CA) for increased throughput using a single GFE as a baseband modem. This type of CA is an important feature in current and next generation systems since it enables baseband processing of tens or hundreds of MHz of bandwidth for high throughput. We are able to perform CA without any other devices and with all the carriers presented at once.

For our experiments, we used NVIDIA's GeForce GTX 680 and 970 GPUs to implement our GFE. The GTX 680 GPU is based on the Kepler architecture, whereas the GTX 970 is based on the newer Maxwell architecture. We have highlighted some of the key specifications and differences in Table 5.1. We used the latest CUDA version 6.5. Our target latency for real-time operation is the 3GPP radio frame length of 10 ms. We used 32-bit complex floating-point precision throughout the experiments. We profiled our kernels on each GPU independently to evaluate the differences in performance.

5.5 Results and Analysis

In our experiments, we measure the performance of 3 kernels: (1) the PFB channelizer kernel, which includes data shuffling, polyphase filtering, and CUFFT operations; (2) the polyphase interpolation kernel; and (3) the TM kernel with NN or LN filtering options. CUFFT, a component in NVIDIA's library of signal processing blocks, is a parallel version of the DFT that is highly optimized for CUDA.

Table 5.1: Summary of relevant GPU specifications.

	GTX 680	GTX 970
compute capability (CC)	3.0	5.2
CUDA cores	1,536	1,664
multi-processors (MP)	8	13
CUDA cores per MP	192	128
GPU clock rate	1.06 GHz	1.25 GHz
memory clock rate	3 GHz	3.5 GHz
memory bus width	256-bit	
maximum texture size (x, y, z)	1D=(65,536), 2D=(65,536, 65,536)	
total amount of global memory	2 GB	4 GB
total constant memory	64 kB	
total shared memory per block	48 kB	
total registers per block	64 kB	

In Table 5.2, we report the resource usage of the different kernels using the GTX 680 as a reference GPU. Here, we show the number of registers used per thread and the amount of SM usage per block in each kernel. Global load and store efficiency measures efficiency in grouping of data in memory — i.e., byte alignment and warp multiples of data to maximize bandwidth. Misaligned accesses of contiguous data or striding causes reduced efficiency and bandwidth. The shared efficiency shows the ratio of SM usage to maximum SM capacity, and similarly, the achieved occupancy

Table 5.2: Resource usage of different kernels in the GFE design.

for GTX 680	shuffle	channelizer	interpolator	TM
registers / thread	8	16	39	8
SM utilization per block	0	4,117 B	504 B	0
global load efficiency	100%	99.8%	82.9%	n/a
global store efficiency	50%	25%	25%	91.4%
shared efficiency	n/a	51.5%	50%	n/a
achieved occupancy	79.5%	90.2%	68%	79.2%

shows the ratio of active warps to the maximum available warps. For more details about these metrics related to GPU implementation, we refer the reader to [2].

As Table 5.2 shows, the load efficiency is near 100% for all of the kernels. Our enforcement of linear read operations helps to achieve this high utilization level. However, due to irregular data access patterns in the polyphase filter structure, the data storage efficiency for the kernels is lower. The kernels containing polyphase filters have relatively low data storage efficiency (approximately 25%). This is improved in our implementation from a level close to 0% in our earlier design [49]. We achieved this improvement by minimizing bank conflicts and using interleaved, complex data write operations, as we have described earlier. Improving global store operations for polyphase filter structure is a good future research direction. Finally, we see that the achieved occupancy is relatively high for all of the kernels. Also, there are no register spills and both polyphase kernels fully utilize FMA operations. Overall, our kernels are highly optimized and achieve high performance, as demonstrated by our profiling.

In terms of resampling accuracy, we compare our ASRC to a rational resampling ratio of $R = \frac{768}{500}$, which resamples a 5 MHz signal to 7.68 MHz — i.e., to twice the desired data rate in our application. Such a rational resampling requires over 3,800 filter taps. We arbitrarily resample UMTS's QPSK waveform from 5 MHz to 7.68 MHz and compute the mean square error (MSE) relative to the rational resampler as a reference. The combined polyphase interpolator and TM-based ASRC had an MSE of 4.11e-4 relative to traditional rational resampling. The difference between NN- or LN-based TM filtering was approximately 1e-5, and LN provided a slightly smaller MSE value than NN, similar to findings in [64]. Therefore, both methods exhibit low error resolution compared to a higher quality rational resampling. We enforced an initial integer interpolation rate of $P = 16$. Even lower error resolution can be achieved if we increase P. However, due to the CUDA Array dimension limit of 65,536 (or 64k) elements for 1D or 2D TM filtering, we limited P to 16.

A novel aspect of this chapter is the ability to process multiple slots or a complete radio frame using 2D TM. This is due to the fact that one UMTS slot contains 3,333.3 samples, however, after interpolation by $P = 16$, we have 53,333 samples. We need to stay under the 64k sample limit of CUDA Array in order to take advantage of the TM filtering option, since samples must be bound to CUDA Array prior to calling the TM kernel. However, since there are 15 slots in a full radio frame of UMTS, we cannot use the 1D TM (CUDA Array) option since it will be limited to a single slot only. We need to process all 15 slots in order to process the full radio frame, which will not fit in a single 1D TM kernel call.

To address this problem, we assign slot numbers along the y-dimension such that in the x-dimension, we still have slot samples. By introducing such a 2D decomposition approach, we can use 2D TM for filtering, while processing the entire radio frame at once. This is possible, since we have another 64k texture elements in the y-dimension, as can be seen in Table 5.1. In this chapter, we aim to process the maximum UMTS bandwidth of 80 MHz or 16 5-MHz wide channels. Therefore, we have used only 240 texture elements in the y-dimension, and we can process as many as 4,369 channels in this architecture.

Also, we note that unlike a typical 2D TM operation, we do not perform a bilinear interpolation. Instead, we perform 1D linear interpolation across the x-direction only — that is, only on the samples within one radio frame of a channel, not across the channels in the y-direction. To summarize, a slot contains 3,333 samples, whereas a frame contains 50,000 samples. Upon interpolation, each slot contains 53,333 samples, whereas a frame contains 800,000 samples for one channel. This highly-interpolated radio frame is then decomposed into a 2D structure, as shown in Figure 5.5. This structure in turn is decimated or resampled at the desired fractional resampling points, Δ automatically using the 2D CUDA Array and TM unit of the GPU.

We now measure the kernel performance of our channelizer, interpolator, and TM ASRC for a single channel as a baseline measurement. The results are summarized in Table 5.3. Here, the channelizer includes shuffle, PFB filtering, and CUFFT

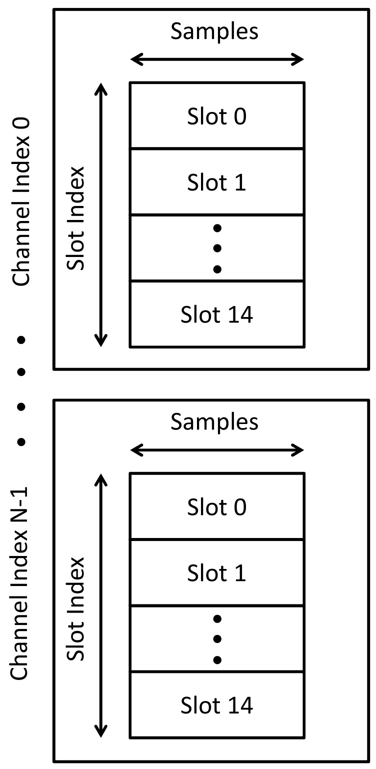

Figure 5.5: Block diagram of multiple slots mapped to a 2D CUDA Array for 2D texture memory.

operations, and we choose to use the LN option in TM. From Table 5.3, we see that all of the kernels exhibit low latency, and in fact, the overall run-time for a single channel is under the UMTS slot time of 0.667 ms. We do not include the effect of memory transfers in these measurements since the data will be processed further in the signal processing chain, and the data is brought back to the CPU for debugging purposes only.

As an additional comparison: at 12 channels, the PFB channelizer presented in [49] ran for 0.734 ms. In contrast, the 12-channel channelizer presented in this

Table 5.3: Single-channel GPU kernel run-time measurement.

	channelizer	interpolator	ASRC	total
GTX 680 time (ms)	0.032	0.373	0.048	0.453
GTX 970 time (ms)	0.041	0.231	0.053	0.325

chapter runs for 0.434 ms and 0.518 ms on the GTX 680 and 970, respectively. This level of performance for our new channelizer is not only less than the slot time, but it is also at least 30–40% faster than the implementation presented in [49] from the optimization we have mentioned earlier.

Next, we increase the number of channels one at a time, and measure the corresponding impact on kernel performance. The results are shown in Figure 5.6, Figure 5.7, Figure 5.8, and Figure 5.9. Although, our goal is to process 16 channels or 80 MHz of BW, we extend our experiments to 20 channels since the modern 4G standard requires as much as 100 MHz of BW. Overall, our GFE runs well under the target latency of a 10 ms even though we are processing multiple channels. Therefore, we expect throughput gain as well. It is interesting also to note the linear relationship between the run-time and the number of channels. Although both GPUs have similar numbers of cores, the GTX 970 provided faster execution times, and more consistent levels of improvement compared to the to the GTX 680, as seen in the figures.

On each of the targeted GPU devices, our improved version of the PFB channelizer completed the channelization process up to 20 channels in approximately 1

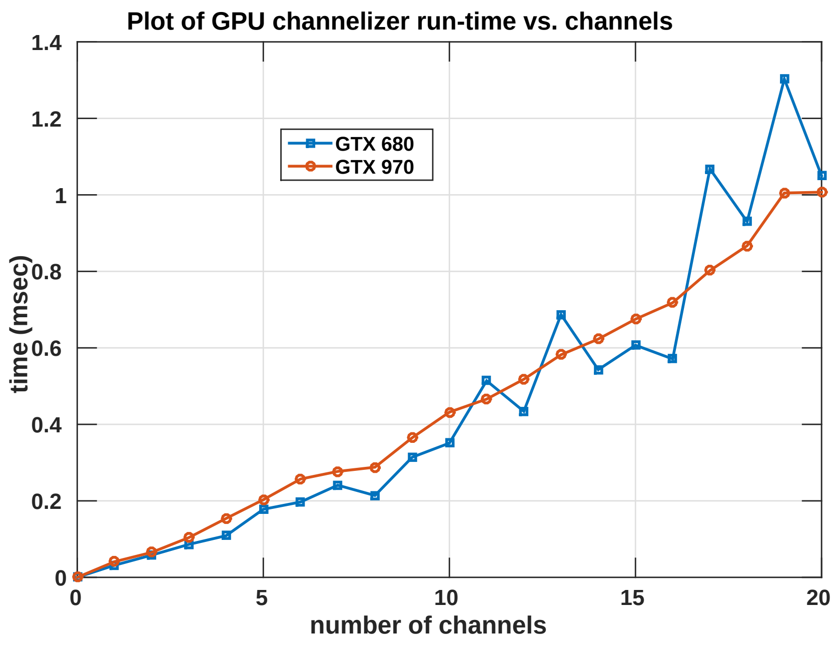

Figure 5.6: Plot of GPU channelizer run-time for an increasing number of channels.

ms. The interpolator kernel was the most compute intensive kernel. Since it produces more outputs than the input data, it requires longer execution time compared to the other kernels. The TM ASRC kernel ran much faster even with the maximum number of channels. This is because of its use of a dedicated hardware unit for computing the resampling points directly, and because the kernel produces a smaller amount of output compared to an interpolator. Without such an optimized implementation, this kernel would be the most compute intensive kernel since it would require brute force computation of fractional resampling points and polynomial approximation. These computations are performed in our TM kernel at no added cost. This represents a distinct advantage over other platforms of using a GPU as a radio.

In Table 5.4, we summarize the run-time and throughput results for our im-

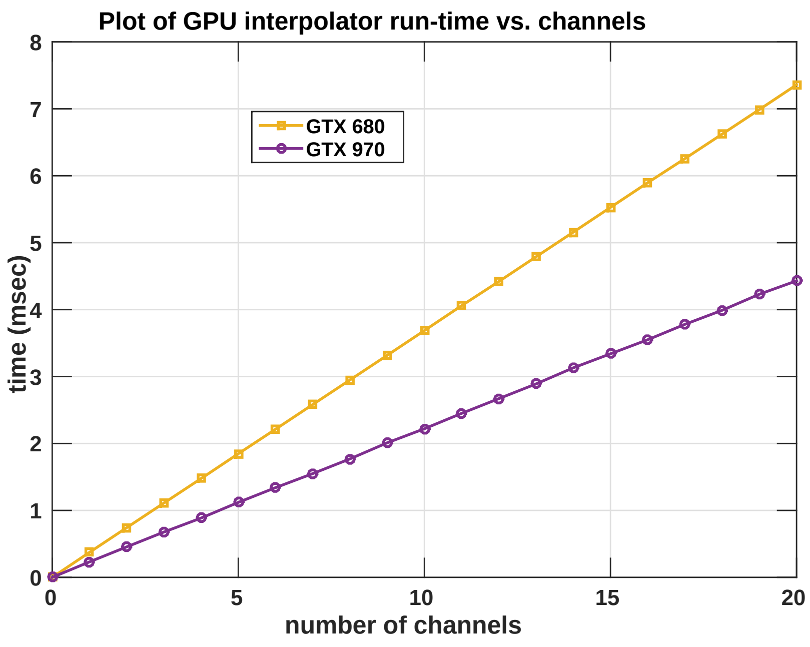

Figure 5.7: Plot of GPU interpolator run-time for an increasing number of channels.

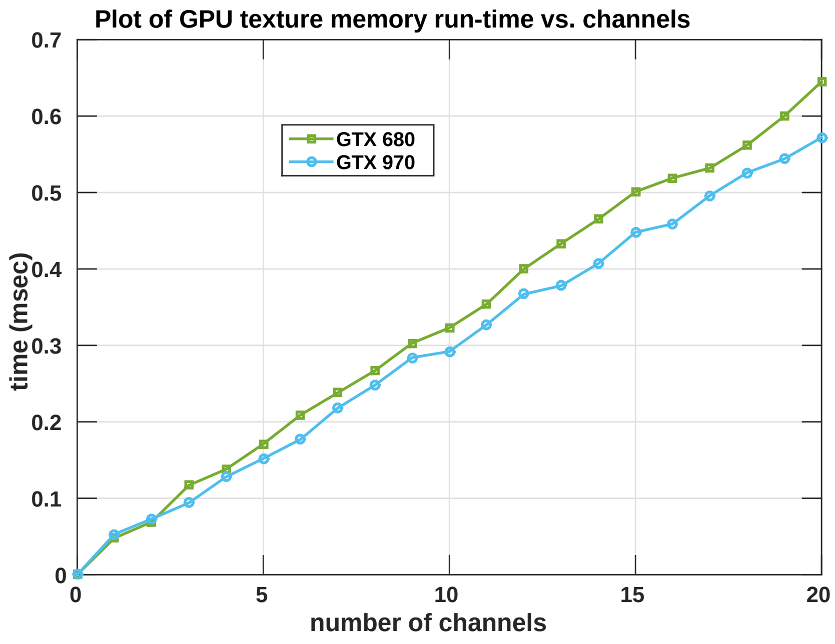

Figure 5.8: Plot of GPU texture memory run-time for an increasing number of channels.

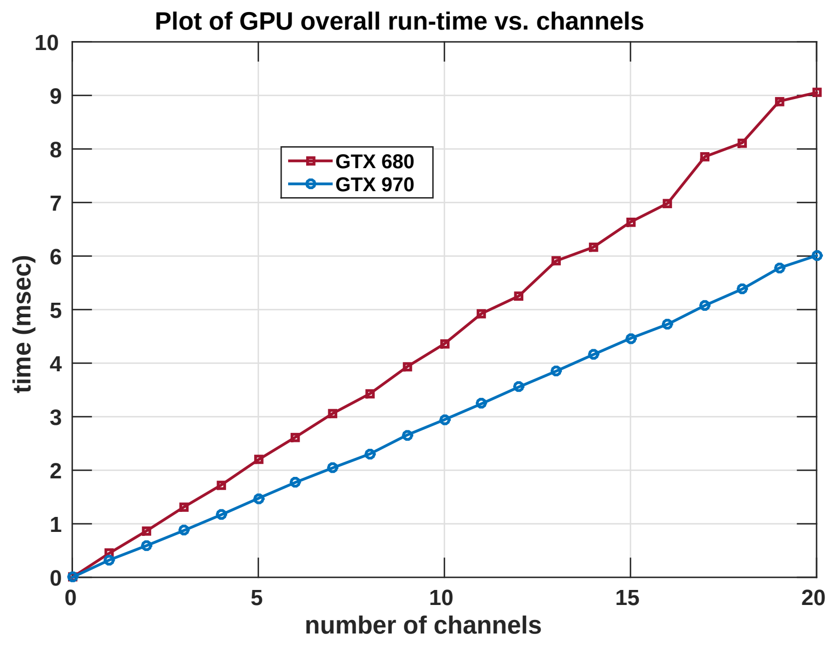

Figure 5.9: Plot of GPU overall run-time for an increasing number of channels.

plementation for processing all 16 UMTS channels. As shown in this table, we can process the entire 16-channel radio frame in less than 7 ms using the GTX 680, and in less than 5 ms using the GTX 970. Thus, for each of the targeted GPU types, our implementation allows for the processing to be completed in less time than a single UMTS radio frame duration. Even when taking memory transfer time into account, the system provides a throughput increase, although there is a longer run-time. Additionally, we note that if HSPA's 2 ms subframe time is the desired latency, then we can process up to 4 and 6 channels simultaneously using the GTX 680 and 970, respectively.

Not only does our developed GFE satisfy the real-time latency constraint, it also achieves massive speedups in throughput on both of the targeted GPU devices. The speedups are calculated based on the 3.84 MHz chip rate of UMTS. This sampling rate is a common sampling rate of 3GPP's UMTS and LTE. Therefore, our architecture design can be extended to other standards as well. In addition, because we are channelizing and processing each channel independently, we can adapt this architecture for spectrum sensing [71–73]. This type of FBMC-based wideband parallel transceiver is an attractive option for spectrum sensing since we can detect the energy of different carriers in parallel in a single instance, instead of sequentially cycling through each carrier.

Finally, because we can process multiple 3GPP channels within a band, we can perform intra-band carrier aggregation (CA), which will increase the throughput further since a composite, wideband channel will be presented at the baseband. With our low-latency GFE, this system is capable of even higher throughput and speedup over the required data rates in UMTS's WCDMA and HSPA. This is also a multi-carrier system that can service simultaneous CA options, which is one of the requirements of the current and next generation communication standards requiring hundreds of MHz of BW [18].

Although our main emphasis here is with UMTS, a band is shared with other standards as well, such as GSM and LTE. Since, multiples of 5 MHz channel spacing are commonly deployed, we can apply our PFB channelization approach up front with mixed standards. In the case where only UMTS channels are desired, other non-COIs can simply be discarded since the information is in the phase of the channelizer, there is no cost or penalty for discarding non-COIs. In addition, our GPU-based ASRC method can readily be adapted to different sampling rates, such as resampling of GSM and LTE channels. Therefore, our GFE is a flexible, low latency, multi-

Table 5.4: 16-channel GFE results (run-time / throughput).

GTX 680	without memory transfer	speedup	with memory transfer	speedup
channelizer	0.571 ms / 2,802.53 MSps	729.71x	3.054 ms / 523.84 MSps	136.43x
interpolator	5.896 ms / 271.42 MSps	70.67x	36.44 ms / 43.92 MSps	11.43x
TM	0.519 ms / 3,081.33 MSps	802.82x	3.851 ms / 415.37 MSps	108.22x
overall	6.986 ms / 229.03 MSps	59.64x	43.34 ms / 36.92 MSps	9.61x
GTX 970	without memory transfer	speedup	with memory transfer	speedup
channelizer	0.718 ms / 2,228.41 MSps	580.31x	2.925 ms / 547.03 MSps	142.45x
interpolator	3.551 ms / 450.25 MSps	117.34x	33.74 ms / 47.43 MSps	12.35x
TM	0.459 ms / 3,480.65 MSps	907.77x	3.721 ms / 430.17 MSps	111.98x
overall	4.727 ms / 338.48 MSps	88.15x	40.38 ms / 39.62 MSps	10.32x

channel, and high throughput system that is capable of processing many channels simultaneously. Overall, we have demonstrated significant performance increase over the traditional, dedicated, fixed-point, hardware approach.

5.6 Summary

In this chapter, we have developed a novel GPU-based multi-channel arbitrary resampler for GPU front-end (GFE) transceiver, particularly as a multi-carrier receiver that is capable of channelizing multiple channels simultaneously and arbitrarily resampling them to any desired rate. Our proposed receiver design reduces run-time latency by eliminating serial control loops and spreading more work across the GPU. Our efficient polyphase filter-based channelizer and interpolator do not require commutators, and the system produces output for all of the channels in parallel.

We exploit unique features of GPUs in new ways to provide dynamic arbitrary sample rate conversion capability via GPU's texture memory. This unconventional approach can be used to accommodate various sampling rates without designing any time-varying filters or calculating fractional resampling points each time. This flexible GFE is capable of delivering high throughput, low latency, and low error resolution using an all-software implementation with floating point precision. Additionally, the developed system supports multi-channel, multi- carrier, and multi-slot or full radio frame operation. Overall, our presented GFE design provides a flexible and cost-effective real-time solution using existing commercial off-the-shelf devices over custom devices.

Chapter 6: A GPU Implementation of a Multi-carrier Multirate Resampler

In this chapter, we propose a novel GPU-based multirate resampler that resamples across multiple channels and carriers. This architecture uses an optimized rational resampler that can fractionally resample input samples to the desired output sample rate. We utilize the multidimensional parallel processing architecture of GPUs to extend the fully-parallel resampling method to resample across different channels and bands. Our approach implements carrier aggregation without a need for additional hardware or specialized architectural support and is able to resample wide bandwidth signals in real time. The resulting design provides high throughput, low latency, and low complexity along with the flexibility of an all-software implementation. We demonstrate the performance of our novel multirate resampler by applying it to resample multiple 4G LTE signals up to 100 MHz. The work of this chapter was presented in [74].

6.1 Introduction

In modern communication systems, a sample rate conversion (SRC) is necessary to accommodate different standards' requirements such as different data rates, system clock rates, etc [21]. A single, fixed reference clock is typically used to generate a common clock speed rather than using multiple clock sources or a tunable clock due to advantages in terms of accuracy, stability, and cost. Therefore, in wireless communications transceiver, some form of SRC is needed to convert a fixed clock to the desired sampling rates. One useful SRC technique is to perform a rational resampling. This method is attractive since it can be realized using a fractional ratio using integer multiples and divisions of the input sampling rate [31].

In addition, modern communication systems require wide bandwidth (BW) to provide high data rates. However, BW is a scarce resource, and is fragmented across the frequency spectrum. Usable segments of BW can be hundreds of MHz apart. In order to meet wide BW requirements in the presence of such scattered segments of available spectrum, modern communication systems deploy various forms of aggregation through multiple carriers. The aggregated carriers can be within the same operating frequency band (intra-band) or different bands (inter-band). Also, the carriers do not have to be contiguous; they can be fragmented within a band. These issues pose significant challenges in modem designs since all of the carriers need to arrive at the baseband.

In this chapter, we propose an efficient rational resampling architecture that is highly optimized for real-time implementation on GPUs, and provides the foundation for effective SRC in the context of multi-carrier transceiver system design. In particular, proposed new SRC design method can readily be applied to multiple channels and carriers, thereby enabling real-time resampling of wide BW signals at the baseband. The resulting architecture exhibits high throughput, low latency, and low complexity while providing the flexibility of an all-software realization.

The remainder of this chapter is organized as follows. First, we discuss the theory and operation of rational resampling. We then introduce a novel GPU-based resampling algorithm, and demonstrate an implementation of this algorithm by targeting an important wireless communication standard.

6.2 Related Work

In this chapter, our objective is GPU-based, real-time processing of multi-channel rational resampling, particularly in the context of software radio applications. Preliminary work on multicore acceleration of polyphase filtering was presented in [44, 56]. A comparison of polyphase filtering throughput among different multicore platforms was presented in [75]. A GPU-based arbitrary SRC (ASRC) was presented in [41]. Another GPU-based ASRC implementation, which incorporated GPU's texture memory, was presented in [64]. Much of the related work for resampling polyphase filters is targeted to FPGA and dedicated VLSI implementations for a single channel or stream of data (e.g., see [22, 67, 68]. However, we emphasize on multi-channel design objective in this chapter.

In hardware implementation targeted to wireless communication systems, performance criteria such as throughput and latency can be met by increasing clock speed, and assigning more resources, such as memories and multipliers. In this chapter, we aim to provide performance that meets relevant real-time constraints by purely using software and efficient algorithm mapping onto non-custom, commercially-available devices. We place particular emphasis on multirate, rational resampler design and apply our resampler design to multiple-channels, which allows simultaneous processing of different carriers. Our proposed architecture is unique in that it realizes carrier aggregation (CA) using a single baseband processor using a GPU and effectively resamples the input channels to the desired rate. This is an attractive option for modern communication systems, which require processing of multiple signals located in different bands.

6.3 GPU-based Multi-carrier Multirate Resampler

Using a polyphase filter structure, we design our GPU-based, fully parallel multirate filter. We combine integer interpolation and integer decimation to achieve a fractional resampling, as we have mentioned earlier. Therefore, we develop a single-filter implementation, and modify the indexing at the filter input and output to achieve the desired sample rate ratio.

First, we design the polyphase filterbank (PFB) structure for our design by decomposing the 1D FIR filter of length N into a 2D structure of $P \times M$, where M is the length of sub-filters. The filter coefficients are then stored in constant memory (CM) in the GPU for fast, read-only broadcasting. Next, we perform interpolation by computing the inner product (IP) between the input samples and the PFB. We read M input samples from GM and store them into SM as a user-enabled cache. Here, SM is beneficial since we access the same M input samples P times to present them to rows of the PFB.

We design our GPU kernel such that each thread is responsible for a single multiply-and-accumulate (MAC) operation. To perform these MAC operations, we instantiate fused-multiply-add (FMA) operations in the GPU for speed and accuracy. Using this approach, the entire interpolation and decimation process is contained within a block of threads. This organization provides isolation with respect to other MAC operations and increases occupancy in the GPU kernel, which in turn increases throughput. This approach to GPU mapping of coupled interpolation and decimation operations is an important aspect of the contribution in this chapter.

Following the interpolation, we select every Q samples and discard the rest in a single stage using a single filter design. A potential disadvantage of using a polyphase filter is that it results in irregular data access patterns (i.e., data is loaded column-wise and subsequently operated row-wise), which is detrimental in a GPU since it conflicts with the objective of coalescing or grouping data when accessing GM. To improve the efficiency of these data accesses related to polyphase filter operation, we introduce a swap buffer prior to writing to GM. This buffer serves to linearize the output samples so that there are no bank conflicts, and hence no serialization when writing results to GM. The resulting samples are byte-aligned in memory, and written in parallel in a single cycle.

To process multiple channels in parallel, we exploit the capabilities of GPUs to perform efficient multidimensional signal processing. In particular, we apply a 3D approach where, the x-direction is used to perform all of the resampling for a given channel, and in the y-direction, we assign different channels. Using our organization along the y dimension, each channel performs resampling independently (from the

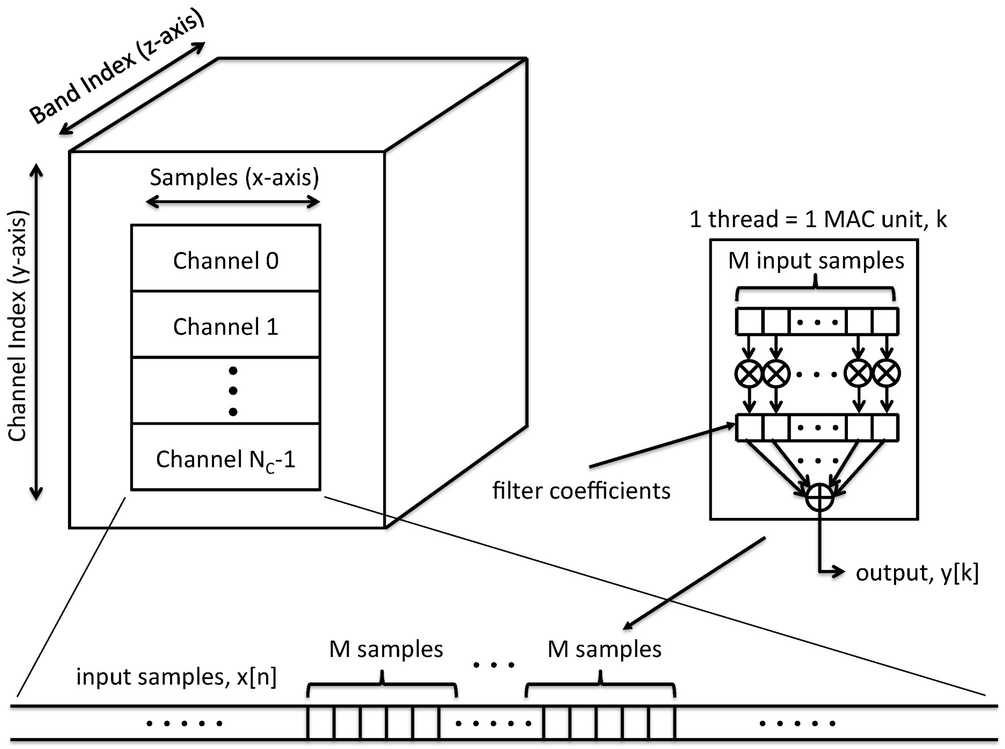

Figure 6.1: Block diagram of multidimensional resampler.

other channels) and simultaneously, utilizing SIMT operations. Finally, in the z-direction, we assign different frequency bands that are outside of the bands that the x- and y-directions are operating in. This organization of processing along the z-direction allows us to efficiently perform simultaneous inter- and intra- band processing for CA in our targeted multichannel implementation framework. Our multidimensional, rational resampling system block diagram is shown in Figure 6.1.

A pseudocode description of our proposed multi-channel, multirate resampling approach is shown in Figure 6.2. This approach efficiently combines multiple waveforms so that they can be resampled to a common sample rate, and combined to a form a larger, contiguous BW at the baseband. Furthermore, in place of mapping different frequency bands in the z-direction, different antenna ports can be mapped as well. With such a modification, our implementation can be adapted to process multiple streams in a MIMO antenna system.

In summary, we have developed a novel GPU-based polyphase resampler that is capable of resampling many channels simultaneously across different frequency bands. Since a single thread is assigned to each output sample, hundreds of thousands of threads are instantiated, which increases workload in the GPU and reduces run-time. Through careful prioritization of data access patterns, our design also ensures high global load and store efficiency, and high GPU occupancy, leading to optimized throughput. In the following section, we demonstrate the significant performance improvements achieved by our new GPU-based polyphase resampler implementation.

6.4 Implementation and Experimental Setup

We demonstrate our proposed GPU-based, multi-channel resampling implementation by targeting the 4G wireless standard, LTE. The radio frame duration of 3GPP's LTE is 10 ms, which is further divided into 10 subframes with each subframe consisting of 2 slots. Therefore, each subframe and each slot has a duration of 1 ms and 0.5 ms, respectively. LTE has a variable BW from 1.4 MHz to 20 MHz, and a sampling rate that is based on a subcarrier spacing of 15 kHz multiplied by the number of DFTs required for modulation. LTE uses CA to form a large composite

BW at the baseband using intra- and inter-frequency bands [76]. Additionally, LTE uses QPSK or QAM for modulation.

We process a single channel containing 10 MHz of LTE BW. This channel is sampled at 25.6 MHz, and needs to be resampled to 15.36 MHz, which is 4 times the common 3GPP sampling rate of 3.84 MHz. Therefore, the resampling ratio $R = \frac{3}{5}$ is required. We design the rational resampling filter using an equiripple FIR filter with 96 taps, which is decomposed into a $P \times M$ polyphase matrix, where $P = 3$ and $M = 32$. This filter has 70 dB of attenuation and 0.3 dB passband ripple.

For our experiments, we use NVIDIA's GeForce GTX 680, 780 Ti, and 970 desktop GPUs to implement our designs. The GTX 680 and 780 Ti GPUs are based on the Kepler architecture, whereas the GTX 970 is based on the newer Maxwell architecture. Our reference GPU in this experiment is the GTX 970; we will compare the results of the other two GPUs to this reference device. We use the latest CUDA driver version 6.5. Our target latency for real-time operation is the 3GPP radio frame duration of 10 ms. We use 32-bit, complex, floating-point precision throughout our experiments.

We are given a 25.6 MHz BW signal that is at a frame duration of 10 ms, and is resampled by $\frac{3}{5}$. Thus, windows of 256,000 successive samples are interpolated by $P = 3$ to 768,000 samples, and then decimated by $Q = 5$ to 153,600 samples to achieve a 15.36 MHz BW. A block in our GPU mapping spans 256 threads in the x-direction and 3 threads in the y-direction, resulting in a total of 768 threads per block (TPB). The resulting kernel provides 1,000 blocks in the x-direction and N_C channels in the y-direction, where N_C represents the number of channels to be

Table 6.1: Single-channel performance comparison of interpolation for different GPUs (run-time / throughput).

GPU	with data transfer	without data transfer
GTX 680	1.967 ms / 260.32 MSps	0.294 ms / 1,742.16 MSps
GTX 780 Ti	2.331 ms / 219.64 MSps	0.177 ms / 2,889.13 MSps
GTX 970	2.356 ms / 217.29 MSps	0.207 ms / 2,474.86 MSps
Tegra K1	16.82 ms / 30.437 MSps	6.532 ms / 78.38 MSps

processed.

6.5 Results and Analysis

We first measure the run-time of our interpolation method for a single channel on different GPUs. The results are shown in Table 6.1. Here, *Sps* refers to samples per second. We also include NVIDIA's embedded GPU, the Tegra K1 (TK1) as a comparison. TK1 runs slower than the discrete GPUs. TK1 is simply included here for comparison purposes; this device is not suitable for a real-time realization of our multi-carrier resampler design, and therefore, it is not included in the rest of our experiments. An embedded GPU such as TK1 provides a significant decrease in power consumption over discrete desktop GPUs, and optimizing our parallel resampler design for embedded GPU is an interesting direction for future work.

The discrete GPUs ran well under 10 ms; in fact, they ran under the slot time duration of 0.5 ms. This is not surprising due to the high parallelism and

Table 6.2: Single-channel performance comparison of resampling for different GPUs (run-time / throughput)

GPU	with data transfer	without data transfer	speed-up
GTX 680	0.881 ms / 580.93 MSps	0.306 ms / 1,671.72 MSps	108.84x
GTX 780 Ti	0.944 ms / 541.89 MSps	0.189 ms / 2,712.78 MSps	176.61x
GTX 970	0.959 ms / 533.51 MSps	0.224 ms / 2,284.73 MSps	148.74x
Tegra K1	8.889 ms / 57.59 MSps	6.319 ms / 81.02 MSps	5.27x

balanced workload distribution across the many GPU cores. Even with the time for data transfer taken into account, the execution time for our implementation was well under the latency constraint. The throughput also increased significantly. In Table 6.2, we compare the execution time results for the resampling operation, and the speed-ups over the resampled rate of 15.36 MSps are shown in the table as well. Overall, our resampling method achieves significant improvement in throughput while achieving very low run-time, and speed-ups over 100 times,

In terms of resource usage, we examine our reference GPU, the GTX 970 in detail. For interpolation, the GM load efficiency is near 100%, meaning the read operation is fully coalesced, since the read operation is linear. However, after polyphase filtering, the output samples are transposed. We address this issue by restructuring the output dimension by using SM and transposing our index to completely avoid bank conflicts in the GM store operation as shown in Figure 6.2. These transformations result in 100% GM store efficiency from 0%.

Coalescing of memory accesses is generally a high priority consideration when optimizing CUDA kernels [40]. However, for resampling, our store efficiency is reduced because of occasional bank conflicts that can arise during decimation — i.e., multiple writes to the same bank can serialize GM store operations and causes delays. Hence, we achieved 73% GM store efficiency during resampling, which is far better than the near 0% efficiency that would result from a conventional polyphase operation. Further improving store efficiency during resampling is a useful direction for future work.

The occupancy calculator based on factors such as block size, register usage, and SM usage, indicates a peak theoretical occupancy of 75%. In comparison, the occupancy achieved by our optimized design is reported as 72%. This result indicates that our kernel is highly optimized with a high level of GPU utilization. Furthermore, there are no register spills and both kernels fully utilize FMA operations.

Finally, we emphasize that a novel aspect of this chapter is to process multiple carriers at the same time. For LTE, CA is used to form a baseband BW up to 100 MHz. Since our design processes a 10 MHz BW, it can process up to 10 channels simultaneously. In Figure 6.3 and 6.4, we show the run-time of 3 GTX GPUs as we increase the number of channels. It is interesting to note the linear relationship of the run-time as the number of channels increase linearly. Here, we do not include memory transfer time (between the CPU and GPU) since the data is left in the GPU for further signal processing. As we can see from these plots, both kernels run under 3 ms, which is well below the target frame time.

6.6 Summary

In this chapter, we presented a novel GPU-based rational resampler that performs resampling on multiple channels and carriers. The resulting architecture is a real-time, GPU implementation that provides high throughput, while achieving low latency. Our polyphase-filter-based resampler requires only a single filter design, and provides integrated interpolation and decimation. Our optimized design addresses some of the disadvantages found in existing approaches to GPU-based polyphase filter implementations by increasing data coalescing and occupancy, and providing improved device utilization. Our design provides the flexibility of a software implementation, and therefore avoids the expense and rigidity of custom hardware, while satisfying the relevant real-time constraints.

```
$tix = threadIdx.x; tiy = threadIdx.y$
$bix = blockIdx.x; biy = blockIdx.y$
$pdx = bix \times blockDim.x + tix; qdx = biy \times SAMPLES\_PER\_ROW + pdx$
$idx = tiy \times blockDim.x + tix; odx = bix \times SAMPLES\_PER\_BLOCK \times P + idx$
$rdx = biy \times SAMPLES\_PER\_FRAME + odx$
if $pdx < SAMPLES\_PER\_ROW$ then
  $SM\_REG[tix + M - 1] = in[qdx]$
  if $tix < M - 1$ then
    $SM\_REG[tix] = in[qdx - M + 1]$
  end if
  $SYNC\_THREADS$
  for $ii = 0$ to $M - 1$ do
    $cplx\_sum += CM\_COEF[ii \times P + tiy] \times SM\_REG[tix + M - 1 - ii]$
  end for
  $SM\_OUT[tix][tiy] = cplx\_sum$
  $new\_tiy = idx/P$
  $new\_tix = idx\%P$
  $new\_cplx\_sum = SM\_OUT[new\_tiy][new\_tix]$
  if $rdx\%Q = 0$ then
    $out[rdx/Q] = new\_cplx\_sum$
  end if
end if
```

Figure 6.2: Pseudocode for GPU-based multi-channel, polyphase resampler

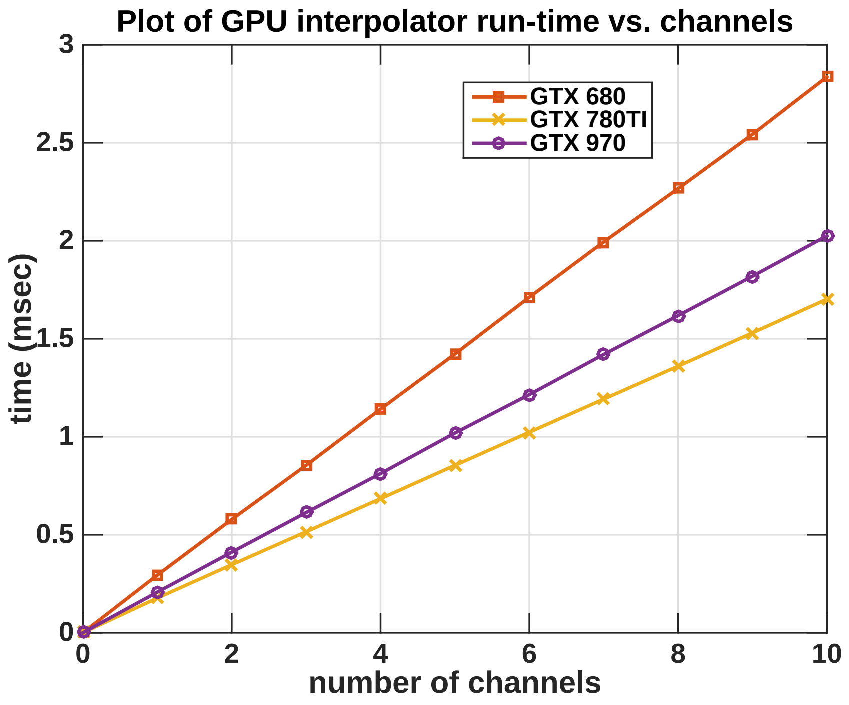

Figure 6.3: Plot of GPU interpolator run-time for increasing number of channels.

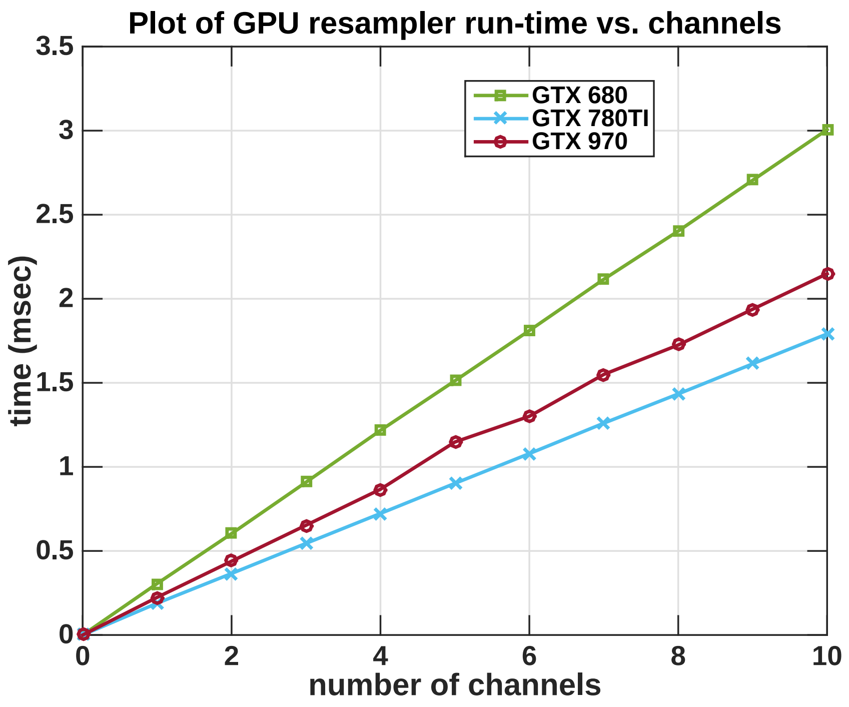

Figure 6.4: Plot of GPU resampler run-time for increasing number of channels.

Chapter 7: GPU Acceleration of Symbol Timing Recovery

In this chapter, we present a novel approach to GPU acceleration of symbol timing recovery. Our approach is targeted to heterogeneous CPU/GPU platforms, and centers on use of polyphase interpolators to detect symbol timing error. Symbol timing recovery is a compute intensive procedure that detects and corrects the timing error in a coherent receiver. We provide optimal sample-time timing recovery using a maximum likelihood (ML) estimator to minimize the timing error. This is an iterative and adaptive system that relies on feedback. Therefore, we present an accelerated implementation using a GPU for timing error detection (TED), thereby enabling fast error detection.

We demonstrate this heterogeneous CPU/GPU implementation by computing a low complexity and low noise matched filter (MF) while simultaneously performing TED. We then compare the performance of the CPU- vs. GPU-based timing recovery for different interpolation rates to minimize the error and improve the detection speed by up to a factor of 35. We further improve the throughput by utilizing GPU optimization and performing block processing, all while maintaining a low sampling rate. The work of this chapter was presented in [44].

7.1 Introduction and Related Work

When data is transmitted over a wireless communication channel, it is corrupted due to various types of noise, such as fading, oscillator drift, frequency and phase offset, receiver thermal noise, etc. At the receiver, it is also immune to noise and symbol jitter in time domain because the transmitter and receiver clocks are not the same. Therefore, a timing recovery subsystem must be able to sample the data at a correct instant and detect its peak for correct symbol timing recovery (STR). Sampling just once at the receiver is ineffective due to noise — i.e., additive white Gaussian noise (AWGN). However, a matched filter (MF) can limit the noise at the receiver and provide a high signal to noise ratio (SNR) sampling point (due to correlation gain). The goal is to obtain best SNR while avoiding inter-symbol interference (ISI). The MF is a time-reversed and delayed version of the transmitted waveform. To maximize SNR for the detection, the demodulator must form inner products between the incoming signal and the reference signal. That means it must time-align the locally generated reference signal with the received signal. Since the inner product is formed in a convolving filter, the demodulator must determine the precise time position to sample the input and output of the filter.

Over the decades, engineers have tried to design and implement clever receivers that not only detect but correct the incoming signal. This was first introduced in the analog domain, however, with the availability of digital integrated circuits, the process was converted over to the digital domain using transformation methods, yet the overall concept and the process remains the same. This process employs a phase-locked loop (PLL), which has 3 major components: 1. a timing error detection (TED) circuit; 2. loop filter (LF) for phase and frequency offset detection; and 3. a controlled oscillator, such as a numerically controlled oscillator (NCO), to advance or retard the timing so that the peak of the incoming signal is matched with the reference signal. There are several widely used methods in TED: the Gardner method [36], Mueller and Muller (M&M) algorithm [77], early-late gate algorithm (ELGA) [4, 32, 78], and maximum likelihood (ML)-based TED [13, 32].

The goal is a TED that yields high SNR, and is resource efficient while maintaining the lowest possible sampling rate (ideally, 1 sample per symbol (spS)), and possibly exploits data independence by using parallelism to speed up the PLL. Therefore, we focus our design using ML-based TED using MF and derivative MF [32]. ML seeks the peak of correlation output using derivative MF (dMF). ELGA is the predecessor in that it essentially finds the derivative by approximation using early, current, and late samples. This provides a relatively low complexity structure for a high performance system, which is critical in terms of designing a resource efficient transceiver such as in FPGA. However, it is compute-intensive: it requires 3 spS and often it also requires high order filters. M&M requires 1 spS but its carrier recovery must be performed before STR. Interpolation techniques for STR have been well discussed in the past (e.g., see [79]). Polyphase interpolator based ML TED was introduced in [32, 35]. This idea was taken further by moving MF into the interpolator, and the resulting structure onto FPGAs in [33]. Then the lowest error resolution was achieved by using an arbitrary resampler instead of a polyphase interpolator in [34] for FPGAs as well. The polyphase filterbank is a 2D matrix structure and its lattice decomposition of multirate filters has been introduced in [12]. While these implementations have made progress towards improved TED, the computational bottlenecks of the algorithm prohibit maximum SNR for low sample rates.

Graphics processing units (GPUs) represent an attractive class of computational resources for applications that can map to it. We recognize the independence among the filterbanks and multiplication between filter coefficients and input samples. We can exploit them using multiple forms of parallelism inside the GPU to speed up the overall filtering operation, which then speeds up the overall error detection because its output is directly responsible for the output and timing error [32]. We propose GPU-based TED for STR. Finally, by driving the LF and NCO (running at 1 spS as derived in [32]), it essentially aligns the reference symbol (matched filtered data) to the received data (same principles as other digital methods and as well as analog methods), and this method works well for this type of data-aided coherence receiver (i.e., phase modulated). It keeps the resampling and realigns the sample to the received sample. With decreased detection time, we can increase the throughput of the system by performing faster locking.

To accommodate the iterative and adaptive nature of PLLs, we present in this work a specific decomposition and mapping of the application onto GPUs. With our careful implementation and the availability of many threads and cores in the GPU, we also perform simultaneous STR over multiple input samples. Instead of the sample-by-sample processing in traditional digital receivers, we enable block processing of multiple symbols simultaneously to improve the throughput even further,

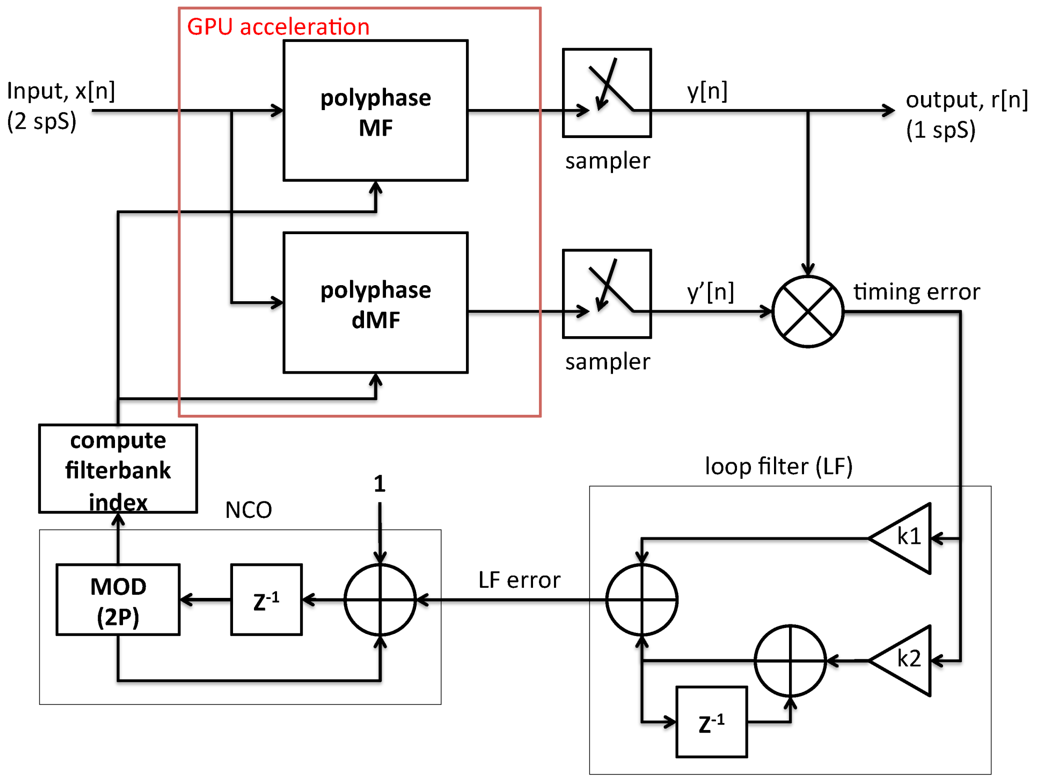

Figure 7.1: Block diagram of ML-based symbol timing recovery.

an attractive option for modern wireless communication systems. The rest of the chapter is organized as follows: we discuss the details of ML- based TED; present our mapping of TED onto GPUs; and present design and implementation details, followed by results analysis and summary comments.

Figure 7.1 shows a block diagram of our targeted communication system throughout the developments of this chapter. We assume that we have a single carrier narrow band system, and that this system has been properly demodulated and downconverted to baseband for STR.

7.2 GPU-based Timing Error Detection

Our design process began with the timing recovery system shown in Figure 7.1, and we first identified the computational bottleneck in this system — i.e., the part that is most arithmetically demanding. The NCO is a sequential system that simply counts up at a certain rate and wraps around after it reaches its peak. We embedded a control circuit in the NCO to scale the output of the LF so that the NCO speeds up or down depending on the error value relative to the peak. The LF is also a sequential system that multiplies the detected timing error by LF gains to track the error over the time. Calculation of gains for TED and LF (K_i and K_p) are well covered in [3, 80], and BPSK timing recovery s-curve calculation is covered in [4]. Our contribution in this work includes mapping such calculations efficiently into GPU implementation, and structuring parallelism within and across the different calculations to maximize performance.

The PLL operates in several modes. During its initial phase or the acquisition phase, it acquires the signal using a wide bandwidth, which in turn allows more noise to enter the loop, but reduces locking time. Once it locks onto a signal it stays in the tracking phase where the bandwidth can be narrowed as much as possible and the loop stays locked as long as the noise level remains stable.

It is important to note that in our system, we fixed our bandwidth to be as narrow as possible going into the loop. Normally, this would cause the loop to take a long time to lock onto the signal, however, with our GPU-based TED, we are able to lock quickly while maintaining this narrowest possible bandwidth in tracking mode, and indeed this is an important novel feature of our GPU-based implementation. This feature provides rapid locking without leaving the system susceptible to larger noise levels across a larger bandwidth.

Switching or changing the bandwidth on the fly is a difficult task but we are able to apply a narrow bandwidth, as described above, and to acquire and lock onto the signal all at the same time. This not only simplifies the design but gives us a smooth tracking curve that is fine tuned over the symbol time. However, the design and implementation of LF and NCO are beyond the scope of this chapter. Therefore, the polyphase MF and dMF, which are based on matrix operations, are the obvious choices to implement on the GPU, and thereby reduce the overall processing time. In a heterogeneous processing fashion, we offload our filter calculations to the GPU and work with sequential subsystems (the LF and NCO) on the CPU.

The MF and dMF are the heart of our targeted design and critical to error detection. The smaller the error or closer to the peak it is the better. Therefore, ideally we want to upsample as much as possible. For serial processors, upsampling heavily (e.g., 1:32 interpolation) is not desirable due to the required resource usage, and such interpolation can require long computation times, and even longer times required to lock onto the peak. An alternative to high interpolation TED is to use an arbitrary resampler, as presented in [13, 34]. However, such a method is complicated to implement. The arbitrary resampler takes interpolation filtering one step further by linearly interpolating between the available output samples of the P-path polyphase interpolator. It yields highly accurate TED without the need for a high P-path interpolator (in such an interpolator the filter is large and indexing

```
for $jj = 0$ to $P - 1$ do
  for $ii = 0$ to $M - 1$ do
    $prod = h[ii \times P + jj] \times r[ii]$
    accum = accum+prod;
  end for
end for
```

Figure 7.2: Iterative MAC operation.

through the filterbanks is slow, resulting in high overhead).

Therefore, a key trade-off is the complexity of the design vs. the resolution of the error or the peak detection. Our objective is to reduce the design complexity while achieving a high interpolation rate. By using a polyphase interpolator to interpolate at a very high rate to achieve arbitrary resampler like performance, and by carefully mapping the filter operations into efficient parallel realizations on the GPU, we achieve this objective through our new approach.

To map the TED onto the targeted GPU architecture, we use warps, shared memories (SMs), and groups (blocks) of multiprocessors (MPs) to optimize utilization of the NVIDIA GTX device. The filter equation has two parts, one for multiply-and-accumulate (MAC) operations to perform the inner product between two vectors — the input array and filter coefficients, and the other for indexing through the filterbanks. A typical polyphase interpolator implementation can be described as shown in Figure 7.2.

Here, h is the filter array, r is an array of input samples, P is the interpolation

```
for $ii = 0$ to $M - 1$ do
    $prod = h[ii \times P + iy] \times r[ii]$
    $accum[iy] = accum[iy] + prod$
end for
```

Figure 7.3: Parallel MAC operation.

rate, and M is the length of a subfilter. Thus, the original filter length is $N = P \times M$. We simply rearrange or reshape this $1 \times N$ filter vector into a $P \times M$ polyphase filter matrix. Instead of performing $O(N)$ operations, the polyphase structure improves this operation to $O(M)$ operations. Due to its 2-dimensional structure, we use double `for`-loops to accomplish this filtering task, which serially indexes through the filter taps and input samples. We utilize multiple forms of parallelism in this structure. Specifically, we parallelize: (1) across the filterbanks (outer loop, `jj` index); (2) across the filter (inner loop, `ii` index); and (3) at a higher level, across the filter and the filterbanks.

When we parallelize across the filterbanks, we exploit the independence of accumulation across the filterbanks. The modified computation structure can be described as shown in Figure 7.3.

Here, we replace `jj` with `iy`, the polyphase filterbank index, and place one filterbank per block in the GPU. Thus, each bank produces one interpolated value or an interpolant. In this version, we improve the performance of the filtering by $O(P)$ operations. Similarly, when we parallelize across the filter (`ii` index) itself, we simply assign one multiply operation to one thread in a block. So we simply

```
$prod = h[ix \times P + iy] \times r[ix]$
$SYNC$
for $kk = 0$ to $M - 1$ do
    $accum = accum + prod[kk]$
end for
```

Figure 7.4: Fully parallel MAC operation.

replace `ii` with `ix`, the thread index of the block. In this case, we improve the filter operation by $O(M)$. The value of M is chosen to match the warp size or 32 threads. We eliminate `for`-loops in the GPU implementation as long as there are no data dependencies, and we can calculate the iterations independently.

Based on this approach, we combine multiple levels of parallelism to parallelize across the entire polyphase filter matrix. The resulting computational structure is shown in Figure 7.4.

In this version, the filter is accessed via thread index `ix` and bank index, `iy`. We use the “sync thread” function in CUDA to synchronize our threads, and ensure that all of the products are available before they are summed. Since we are summing across a relatively small number of threads, it is not necessary to perform further reduction of the accumulator part. Therefore, we sum the products over the threads using a simple `for`-loop, as shown in Figure 7.4.

Using this approach, the performance of the complete GPU-based filtering improves by $O(M \times P)$ operations compared to the original version (Figure 7.2), which is a considerable speed up. Furthermore, our accelerated implementation achieves instant error and output calculation without reducing this high degree of speedup.

In order to optimize the GPU implementation for our experiments, we choose the number of threads per block (TPB) to be a multiple of the warp size to avoid wasting bandwidth and facilitate coalescing. Each thread is assigned a lightweight operation such as multiplication for both filters. The interpolation rate is chosen so that all MPs are uniformly loaded — i.e., the same number of blocks is launched on each MP, and also the amount of work of interest per block is the same and provides more consistent results from run to run, which allows a high interpolation rate and higher utilization of blocks on the GPU. The speedup in our implementation is achieved from invoking more GPU blocks, since there are many GPU blocks compared to threads, assuming the threads are kept busy enough (at least 64 TPB).

The output of the STR comes from the MF directly. Therefore, the higher the value of P (the number of polyphase paths), the more accurate the output will be. Furthermore, since the results are based on the actual value rather than the sign, as in [36], it is critical that we align the sample to the peak as close as possible to yield a high SNR. However, increasing P does not always yield a better result, as there is a limit on how far we can interpolate and at some point it does not return lower error and potentially it can slow down the locking time because of the large number of interpolants that must be processed.

In the following section, we experiment with different values of M and P to find where the GPU performs best. We strive for 50% occupancy, which amounts to 256 TPB in the targeted GPU. Our design is structured to utilize SM as much as possible and use constant memory (CM) for read only data, such as filter coefficients, for faster cached access. We minimize register spills to local memory by minimizing local variable declarations and keeping the local array (e.g., product vectors) in the SM as much as possible, and avoiding bank conflicts within SM.

7.3 Implementation and Experimental Setup

We model and simulate our entire design using MATLAB, and then develop optimized implementations in C and CUDA targeted to the CPU and GPU, respectively. For our experiments, a BPSK signal is generated and pulse shaped at 2 spS using a root-raised-cosine (RRC) filter (with a roll-off factor of 0.5). To emulate a burst transmission or a data packet, we choose our data to be 2,000 symbols or 4,000 samples after pulse shaping. Typically, the system requires 500-700 symbols to lock, so it is reasonable to validate the STR operation with 2,000 symbols. AWGN is then added to the transmitted data to emulate the timing jitter in the receiver. It is assumed that the data has been properly modulated then demodulated to the baseband and downsampled to 2 spS immediately going into the timing recovery loop. The matched filter is also an RRC filter (with a roll-off factor of 0.5), and chosen to be of 864 taps, which reshapes it to give a $P \times M$ matrix with size 27×32. Therefore, the number of filterbanks or interpolation rate is 27 and each filterbank has 32 taps. The interpolation rate is varied in order to profile the performance of our system, and help tune the system for maximum performance. The CPU used in our experiments is a dual core Intel Xeon 3.0 GHz CPU, and the GPU is an

NVIDIA GTX 260.

In our design of the STR, we have improved the NCO such that the zone test shown in [32] is not required. The zone test is used in [32] because given 2 spS, even and odd samples are continuously arriving, and the system needs to determine where the peak is. Therefore, tests are performed to determine the decision sample for each input sample. In our modified approach, we streamlined the NCO to simply count up to the total number of samples per symbol. For example, given $P = 27$ and input data at 2 spS, we simply count up to 54 samples per symbol every time. This method eliminates the need for the zone test and significantly reduces the design complexity.

From these 54 samples, only one sample is selected to be used as an error and output of the system. Therefore, running at the lowest possible sample rate is important. The scale factor is given by

$$K_v = \frac{2\pi}{S \times P}, \tag{7.1}$$

where S is the pulse shape rate, and (as defined earlier), P is the interpolation rate.

When the LF error is scaled with this value, the control loop will update the corresponding filterbank index, and the updated index will be used to select the peak on the next symbol. A sample plot of the timing error and the corresponding polyphase bank index is shown in Figure 7.5.

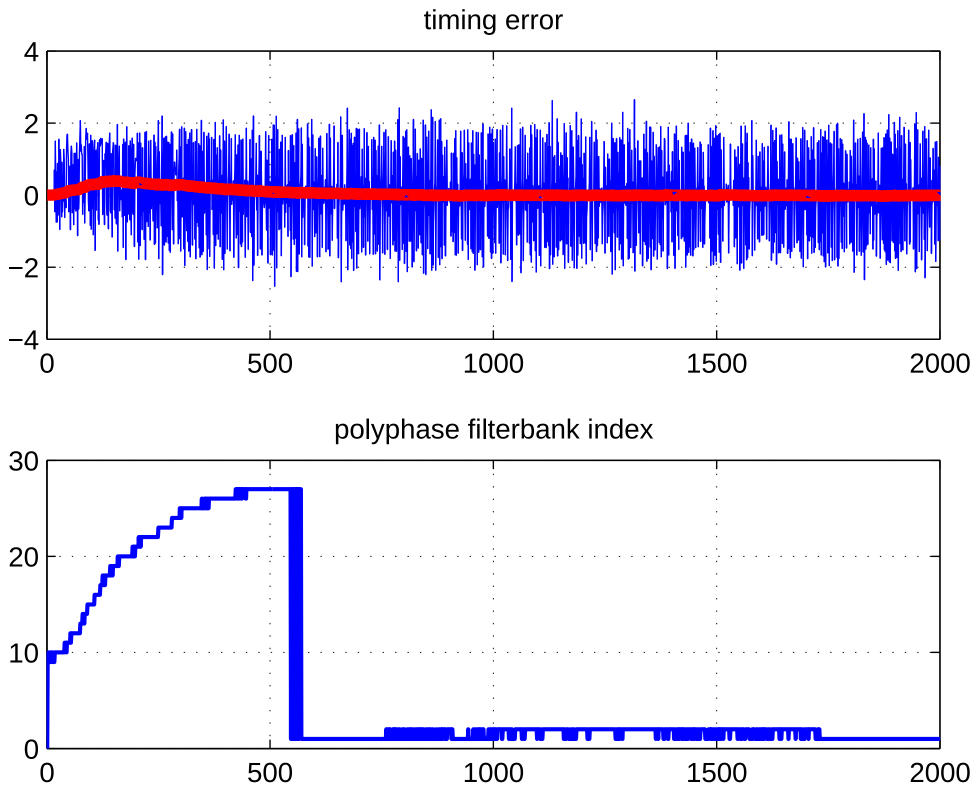

Figure 7.5: An example plot of timing error and corresponding polyphase filterbank index for SNR of 10 dB.

7.3.1 Sequential Symbol Timing Recovery

In Section 7.2, we discuss how we exploit multiple forms of parallelism found in our TED. We first parallelize across filterbanks, followed by parallelization within individual filters. For the initial implementation, we use $P = 27$ and $M = 32$. Each filterbank is assigned to a block in the GPU, where each block has 32 threads assigned to a 32-tap filtering operation. The interpolated values of the input sample are obtained as the matrix-vector product

$$\overline{p} = \overline{\overline{H}} \times \overline{r}, \tag{7.2}$$

where $\overline{\overline{H}}$ is the polyphase filter matrix (dimensions of $P \times M$), $\overline{r}$ is the input array (dimensions of $M \times 1$), and $\overline{p}$ (dimensions of $P \times 1$) gives the interpolated values of the input sample. This process is then repeated for both the MF and dMF to give us filtered results in real time.

Polyphase filtering already gives us reduced multiplications due to its use of filterbanks — given our filter size of $27 \times 32 = 864$ filter taps, we only have to perform 32 multiplications, giving us a workload savings of 96.3%. In addition, we parallelize across the polyphase filtering operations, providing significant savings in terms of computation time.

We vary the interpolation rate P to be multiples of MPs (i.e., multiples of the number of multiprocessors per core). In particular, we employ $P = 27, 54, 81, 108$. We find experimentally that rates that are not multiples of MPs cause a reduction in performance. This type of high interpolation is not desirable in typical FPGA or CPU devices, due to the large number of multipliers, and the large amount of time and memory required. However, it maps efficiently into GPU implementation, and therefore demonstrates an important kind of processing in which GPUs are especially well suited to communication system development.

We transfer data back to the CPU from the GPU every time the TED block is called. Due to the sequential and recursive nature of the PLL and NCO, they are not well suited for GPU acceleration, and thus we incur the data transfer overhead required to perform the PLL and NCO computations on the CPU.

7.3.2 Simultaneous Multi-Symbol Timing Recovery

So far in the chapter, we have presented a one-to-one mapping of a sequential filter indexing into matrix operations by unrolling the loops across the polyphase filter matrix. However, this is still a sequential and iterative system that needs to be updated on a sample-by-sample basis. To utilize more banks and threads per kernel launch on the targeted GPU, we recognize that the input samples do not have to be processed sequentially to produce interpolated outputs. Instead, input samples can be interpolated independently on the GPU using the same kernel, while the PLL is updated sequentially as usual on the CPU. Additionally, we apply block processing on the input samples, which significantly improves throughput and minimizes data transfer overhead.

With this new grouping of input data samples, based on a block processing configuration, we introduce the notion of sub-blocks and sub-thread indexing within a single block. Each sub-block is responsible for a single set of $M = 32$ input words. Therefore, a total of $(M + K - 1)$ samples are stored in SM. Here, K is the number of input samples we wish to process and $(M - 1)$ gives the number of previous words to process. In our case, $K = 8$, so there are 8 independent processing subsystems spanning 32 input samples each, which results in 256 active threads per block, an optimum occupancy level for the targeted GPU.

Since we are processing 8 input samples or 4 symbols per kernel launch, our

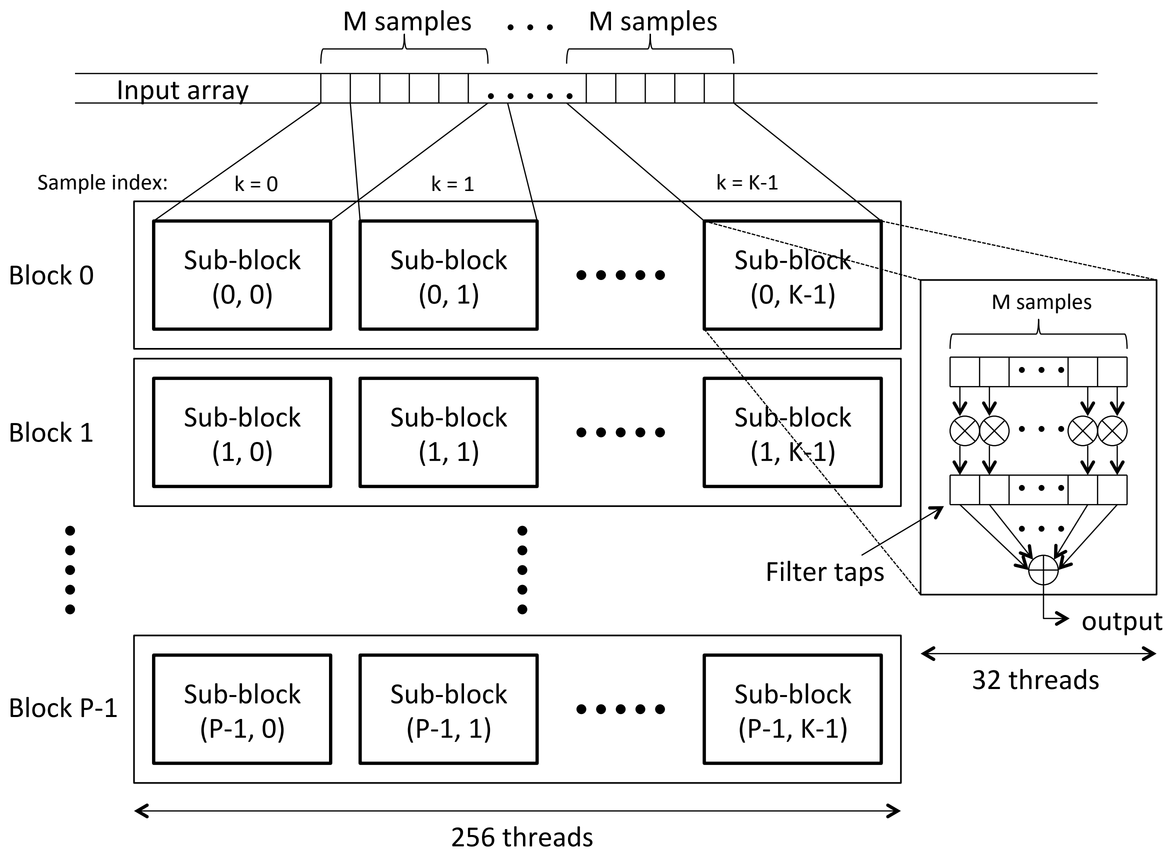

Figure 7.6: Organization of 256 threads to handle eight 32-input words at a time for filtering inside the GPU. ($M + K - 1$ samples are loaded onto the shared memory, where $M = 32$ (a warp) and $K = 8$ (the number of input samples for block processing).

throughput also increases by a factor of 4. Furthermore, we also achieve a reduction in memory transfer bandwidth by a factor of 8. This is achieved because we do not have to transfer the interpolated data back to the CPU for every sample, and our rate for initiating such transfers is reduced by a factor of 8. In addition, the filter coefficients are stored in CM for fast read-only access, and product vectors and accumulation registers are stored in SM for fast read-and-write operations. Our approach to sub-grouping (sub-block organization) is shown in Figure 7.6.

In this adaptive communication system design, it is not obvious how to process multiple input samples simultaneously and still perform iterative updates. However, with our TED operating in the GPU, we can interpolate many input samples all at once, while the CPU updates the loop as usual using the LF and NCO. Given an error value, the LF will be updated and the NCO will continue to add the detected error to its count and traverse along the interpolated samples of the symbol. As the NCO traverses from symbol to symbol, new errors are detected and updated accordingly in the CPU. Regardless of whether input samples are processed one at a time or in groups, the architecture developed in this section provides a significant advantage in that it allows for highly optimized block processing of the data. This results in enhanced real time communication system performance, as we demonstrate experimentally in the next section.

7.4 Results and Analysis

In this chapter, we have presented 5 different TED implementations using CPU and GPU devices. In the CPU, we used double `for`-loops to sequentially index through the filter matrices, whereas in the GPU, we exploited multiple levels of parallelism. These levels of parallelism and their associated implementations are denoted as: (P1) across the filterbank (y-direction); (P2) across the filter (x-direction); (P3) across the entire filterbank matrix (both x and y-directions); and (P4) simultaneous filtering using block processing of the input. Figure 7.7 compares the performance of these different TED designs for different values of the interpola-

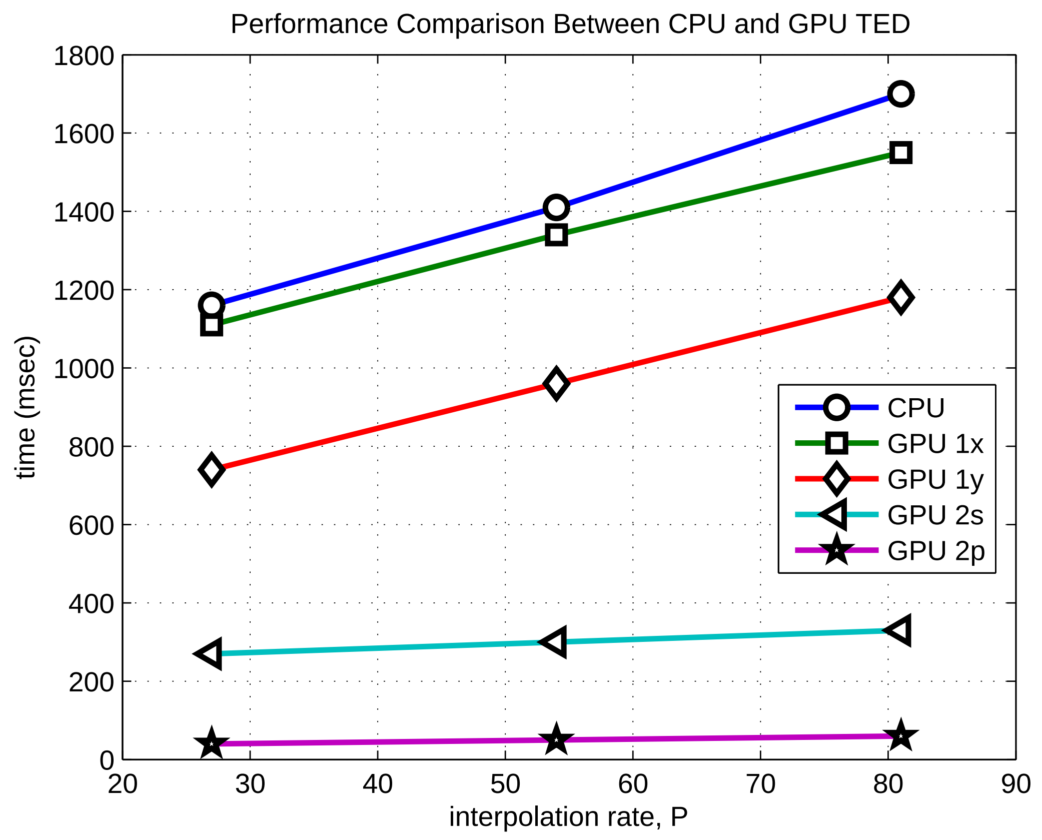

Figure 7.7: Comparison of different TED designs.

tion rate P. Different trade-offs between the interpolation rate and execution time are shown for the CPU-only implementation ("CPU"), and implementations P1-P4, as defined above.

The speedup times (GPU vs. CPU) are summarized in Table 7.1 and 7.2. In this experiment, we used a single GPU version of TED, which was executed in the block processing mode, and with the following additional implementation characteristics: 256 TPB, register ratio of 50% ($8,192/16,384$ or 7 registers per thread), SM ratio of 62.5% ($10,240/16,384$ or 2,300 bytes per block), active blocks per SM of 4:8, and active threads per SM of 1024:1024. Furthermore, none of the

Table 7.1: Achieved speedup for GPU-based implementation of TED.

	P = 27	P = 54	P = 81
Achieved occupancy (%)	25%	50%	75%
overall throughput (GB/s)	2.86	3.35	3.42
CPU TED time (msec)	1,160	1,400	1,690
GPU TED time (msec)	40	40	50
TED speedup	29x	35x	33.8x

interpolation rates (P values) that we experimented with exhibited occupancy as a limiting factor. Only the grid sizes (i.e., the numbers of blocks) or P values were changed in these experiments on overall achieved acceleration.

When we increased P, we observed increasing levels of performance until we reached $P = 108$. At this point, performance began to degrade even though an occupancy level of 100% was reached. At $P = 108$, the overall throughput of the global memory decreased to 3.34 GB/s, most likely due to saturation effects related to the amount of interpolated data that needed to be transferred. As expected, higher occupancy does not necessarily mean higher performance, and our experiments helped to quantify at what point this kind of saturation occurs for our GPU-based TED implementation. Our largest performance gain was achieved with $P = 54$, and an occupancy level of 50%.

Finally, we compared the overall speedup of the STR loop between CPU-based and GPU-based implementations. In this experiment, we used the block processing

Table 7.2: Comparison of the overall speedup for the STR loop between CPU- and GPU-based implementations.

	P = 27	P = 54	P = 81
CPU only (sec)	1.63	2.22	2.85
GPU only (sec)	0.52	0.85	1.19
Overall speedup	3.13x	2.61x	2.39x

version of the GPU TED to maximize the speedup. The results are summarized in Table 7.2.

7.5 Summary

In this chapter, we use a coherent synchronization technique to explore ways to improve the performance of data-aided symbol timing recovery (STR) for a digital receiver. Our targeted STR system is a sequential, adaptive system that must accurately time incoming digital communication symbols under stringent real time constraints. Our goal is to approach the optimal sampling peak as closely as possible while minimizing the error without using filtering that is excessively complex. We use maximum likelihood (ML) based timing error detection (TED) to interpolate the data at a high resolution, and minimize the timing error or detect the symbol peak.

Although we use already streamlined polyphase filterbanks to perform interpolation, the filterbanks create large computational loads due to the high orders of the filters involved. Therefore, we use a graphics processing unit (GPU) to accelerate the operation of TED. Our GPU-based TED enables instant error detection, narrow loop filter (LF) bandwidth (i.e., low input noise) with faster lock, low complexity, and high signal to noise ratio (SNR) with increased throughput. We further improve the throughput by introducing block processing across the input stream. In addition, we simplify our numerically controlled oscillator (NCO) design by performing one update per symbol instead of testing for multiple zones within a symbol.

Our experimental results demonstrate that our new design methods for real-time STR map efficiently into GPU-based implementation, and we provide analysis to quantify some of the key trade-offs involved in this kind of implementation. Building on our proposed STR implementation techniques to develop and optimize complete GPU-based transceiver systems is a useful area for future work.

Chapter 8: Conclusion and Future Work

In this thesis, we have generalized a wideband transceiver implementation using graphics processing units (GPUs), particularly as a parallel front-end transceiver. We presented parallel processing techniques for important signal processing algorithms in real-time software-defined radio applications. Our proposed front-end transceiver architecture can be generalized into four main parts: 1. GPU front-end transceiver implementation, 2. wideband channelization, 3. resampling, and 4. synchronization.

We are presented with discrete-time signals immediately after the sampling circuit in the radio frequency (RF) front-end. The transmitted original signal is corrupted and we must recover it from the channel impairments. A baseband front-end receiver must be able to separate the channel-of-interest (COI) from the incoming signals, i.e., from other users, interferences, and noise. The front-end receiver has a critical task of channelization or separation of the COI, followed by a resampling to match the sampling rate to the desired data rate, and synchronization to match the received signal to the transmitted signal. Following the front-end receiver, the signal goes through further signal processing, such as channel and source decoding.

Wideband channelization is the first stage of our generalized front-end (GFE) processing approach, where we are given a wideband signal (i.e., tens or hundreds of MHz wide) and we demultiplex the channels and select the COI. For this purpose, we have developed a high throughput, low latency, fully parallel channelizer using a polyphase filter bank structure to separate multiple channels simultaneously using an efficient architecture. We have derived several options for channelization using GPUs and demonstrated their performance in real-time applications of wireless cellular standards.

Following channelization, we need to resample to some desired sample rate, since the system clock can be different from the desired sampling rate. We have demonstrated a series of progressively more efficient options for resampling using a sequence of novel implementation techniques. We first demonstrated an arbitrary sample rate conversion (ASRC) system using a fully parallel polyphase filter structure. We then improved the performance of this system by utilizing GPU texture memory (TM) to automatically produce the output signal at the desired arbitrary fractional sampling rate. We improved the design further by enhancing the resolution by interpolating prior to TM resampling. We implemented a fully parallel polyphase interpolator and discrete Fourier transform (DFT) interpolation subsystem to achieve this task. We then took the concept further by resampling across multiple dimensions. We achieved this by exploiting support for multidimensional signal processing structures in GPUs. The resulting system is able to resample within and across relevant frequency bands simultaneously. We demonstrated the performance of our implementations by applying them to wireless cellular standards, and assessing their real-time performance in terms of throughput and latency.

Finally, following resampling, the desired COI must be synchronized in terms of time, frequency, and phase in order to recover the transmitted signal successfully. We investigated one of the most demanding synchronization techniques, symbol timing recovery. We implemented a maximum likelihood estimator using a polyphase filter structure for timing error detection (TED). The estimated timing error was then filtered using a feedback loop structure. The GPU was used to accelerate this TED process by utilizing fully parallel filter operation to calculate the error efficiently. The resulting architecture accelerated overall system performance significantly, and demonstrated large performance improvement compared to a reference serial system.

In the remainder of this chapter, we summarize our contributions to different areas of wideband transceiver implementation, and we outline useful directions for future work.

8.1 GPU Front-end Transceiver

In Chapter 3, we introduced the notion of a GPU-based radio. Specifically, we focused on applying GPU technology to front-end radio design. We designed a wideband receiver using a channelizer to filter, downconvert, and downsample the COI. We included functionality to perform further resampling if the sampling rate at the channelizer output does not meet the desired rate (typically 2–4 times the data rate).

A major contribution in this chapter is utilizing GPU technology to accelerate front-end radio computations, such as filtering and transforms, through software solutions. The resulting architecture exceeds the throughput requirement and reduces the latency cost for real-time operation. A novel aspect of this chapter is our implementation of an interpolating polyphase channelizer to further resample as needed at the output of the channelizer. Following the channelizer, two stages of multirate filtering were employed to perform arbitrary resampling. This subsystem resampled the output of the channelizer to the proper sampling rate, which is fractionally related to the original rate. Even with inclusion of the "post-channelization" resampling subsystem, the GFE architecture presented in this chapter provides increased throughput and reduced latency, and satisfies the relevant real-time constraints.

8.2 Wideband Channelization

In Chapter 4, we presented an improved channelizer design. In this design, we further parallelized the channelization process by unrolling loop iterations across the input samples rather than across the filter coefficients, as was done in Chapter 3. This resulted in significantly larger GPU occupancy, which lead to an increase in throughput and further reduced the latency. We also applied an approach of performing multiply-and-accumulate (MAC) operations using a single thread per operation, which isolated the work of the threads, and made the processing simpler and faster.

Additionally, we developed methods for implementing polyphase filter structures in GPUs by transposing the input data to enforce global load coalescing and caching. This resulted in 100% load efficiency compared to 50% or less from non-optimized structures. This simple transformation further enhanced the development of our channelizer.

Using the approaches presented in this chapter, the output of the GPU channelizer provides all of the desired channels simultaneously. Furthermore, no performance penalty or additional computation is incurred to provide such simultaneous production of the parallel output streams.

We demonstrated the implementation techniques developed in this chapter on a relevant wireless communication standard, and showed that the requirements of this standard can be exceeded using these techniques.

8.3 Multi-channel Arbitrary Resampling

In Chapter 5, we developed GPU-based methods for efficient, arbitrary resampling across multiple wireless communications channels. Arbitrary resampling enables the system to adapt to various sampling rates without changing the system clock rate. However, the process is computationally intensive and can consume a large amount of resources. In this chapter, we introduced a simple yet effective resampling method for GPU-based transceiver implementation.

We revisited our fully parallel arbitrary resampler introduced in Chapter 3 and improved its performance significantly by utilizing a special hardware unit in GPUs called texture memory (TM). TM has a dedicated filtering circuit to perform nearest neighborhood and linear interpolation without computing the fractional resampling points manually. This feature allowed us to take further advantage of GPU capabilities for optimized, software-based ASRC implementation. The resulting architecture provides a low complexity and low latency arbitrary resampler.

To further increase the resampling accuracy, we proposed an integer interpolation prior to applying the TM filtering operation. To achieve this objective, we implemented two different integer interpolations — using time and frequency domain filtering. We introduced an efficient polyphase interpolator and DFT interpolator using a GPU. Our optimized polyphase interpolator is a multi-channel interpolator capable of interpolating across multiple channels, e.g., at the output of a channelizer. We optimized our GPU implementation to improve the efficiency of global load and store operations. In the frequency domain, we utilized a highly optimized, GPU-targeted, DFT library routine called CUFFT to perform integer interpolation.

Collectively, the DFT-based approache presented in this chapter provide a novel ASRC implementation that does not require a filter design process. Furthermore, the implementation is fully software-based through efficient use of GPU technology. We discussed advantages and disadvantages of each of the filtering alternatives considered in this chapter. We again validated the utility of our proposed implementation techniques in the context of practical wireless communication standards.

8.4 Multirate Filtering for Multi-carrier Systems

In Chapter 6, we demonstrated an alternate resampling technique to the ASRC approaches presented in earlier parts of the thesis. More specifically, we presented a common multirate filtering method for rational resampling. This method implements a fixed ratio resampler made up of an interpolation (upsampling) and a decimation (downsampling) to produce a fractional ratio. We combined the two operations into a single stage using a single filter design and an indexing scheme at the input and output to control the resampling rate. This efficient filtering approach is useful in many applications where dynamically-variable, arbitrary resampling is not required — for example, when we know the conversion ratio in advance. We take this concept further by applying the filtering across multiple channels in the same frequency band and then to multiple frequency bands that may be hundreds of MHz apart. This enables carrier aggregation (CA) and resampling of composite wideband signals at the baseband. CA is one of the required features in modern communications systems in order to increase communications throughput.

In our experiments in this chapter, we showed that our GPU-based multirate, multi-dimensional resampling architecture exhibits high throughput, low latency, and low complexity.

8.5 Synchronization

In Chapter 7, we focused on efficient design and implementation of synchronization in the receiver. Synchronization is a critical component of the receiver, where channel impairments are detected, estimated, and corrected in order to successfully recover the transmitted signal. Without synchronization, the other receiver processes would not work properly, leading to high error rates.

In this chapter, we examined the most computationally intensive component of synchronization, symbol timing error detection (TED) and recovery. TED is a critical component of symbol timing recovery since it is responsible for accurately measuring the timing offset. We accelerated this algorithm by using the GPU to perform fully parallel filtering. We incorporated GPU-accelerated TED into a feedback-based design and in conjunction with serial processing in a CPU. We were able to speed up the overall system significantly, as shown by the experiments presented in Chapter 7.

8.6 Future Work

This thesis has examined wideband transceiver design by generalizing major front-end components — the channelizer, resampler, and synchronizer. Future wireless communication systems demand high bandwidth and agile spectrum sharing and deployment. Our application of commercially-available, programmable, GPU devices for real-time signal processing is attractive for helping to satisfy these future needs. GPUs can potentially consume lower power through extensive parallel execution of operations while operating at lower clock speeds. GPUs also have thousands of cores and even more threads available for lightweight signal processing operations. By carefully utilizing the memory hierarchy of GPUs and distributing work across the devices to increase occupancy, which leads to high throughput and reduced latency, communication system designers can employ GPUs as attractive platforms to be used as baseband modems. GPUs can work with dedicated hardware modem subsystems or potentially provide the entire signal processing suite for modem operation.

GPUs are attractive for multidimensional image processing as well as one-dimensional signal processing due to the high levels of data parallelism found in both classes of applications. In addition, GPU implementations are software-based, which is a distinct advantage — in terms of flexibility, design, and verification costs — over hardware that is configured via firmware. Additionally, GPUs are cost-effective due to their large production volumes and their relevance across diverse application domains.

In relation to our overall theme of designing and implementing baseband modems using GPUs, useful directions for potential future work using GPU-based radios includes spectrum sensing using filterbanks, and a complete wideband channelizer radio including synchronization, and multidimensional radio for CA. Before we can demodulate and synchronize the signal, we must detect the presence of a signal-of-interest (SOI). An SOI can be anywhere in the frequency spectrum, and we can apply our fully parallel filter bank approach to analyze the spectrum and sense the active SOI. A GPU approach is promising for this kind of functionality by use of a polyphase channelizer to decompose the incoming (wideband) signal into smaller channels and examine the derived channels in parallel. This approach can provide fast identification of SOIs within such a wide band. Such a topic is an active research topic in cognitive radio and spectrum sensing. Applying fast detection using GPU-based spectrum sensing can enable more effective detection. Combining such a method with machine learning techniques, such as deep neural networks, is another interesting future research direction. Such an approach help to more fully realize efficient spectrum sharing and secure communication systems.

Currently, orthogonal frequency division multiplexing (OFDM) is a popular option to implement wideband transceivers using DFTs. However, OFDM has large spectral leakage at the band edges. This problem can be addressed by applying a polyphase filter prior to the DFT operation to filter the sidelobes. We have discussed such an approach throughout the thesis and have demonstrated a fully parallel polyphase channelizer transceiver. We have also demonstrated its efficient mapping onto GPUs. Once the polyphase channelizer is used as a transceiver, we need to detect, estimate, and correct channel impairments such as frequency, phase, and timing offset. We examined timing recovery schemes for a narrowband signal. A useful direction for future work is extension of these schemes to wideband signals using channelizers to correctly synchronize the SOIs. We expect that this work would require a new architecture and could potentially introduce a new domain for future receiver design compared to the feedback-dominant architectures that are currently used in hardware modems.

An consistent observation from the developments of this thesis is that the concept of GPU-based parallel transceiver architectures has significant potential as a framework for future modem design. This thesis represents various contributions in the evolution of this framework.

Appendix A: Sample CUDA Codes

In this appendix, we provide selected CUDA kernel codes that we developed, and that were employed in the experiments reported on in this thesis.

A.1 Polyphase Channelizer

Here, we provide CUDA code from the polyphase channelizer implementation used in Chapter 4. This code computes a polyphase filter bank operation prior to calling the CUFFT kernel.

```
__global__ void gpu_PFB (float *d_x_real, float *d_x_imag,
                         cufftComplex *d_fft_input}
{
    // LOCAL INDEX
    int ii = 0;
    int tix = threadIdx.x;
    int bix = blockIdx.x;
    int biy = blockIdx.y;
    int pdx = blockIdx.x*blockDim.x+threadIdx.x;
    int idx = biy*NUM_SAMPLES + pdx;
    int odx = pdx*PFB_Q + biy;

    int rdx = pdx*PFB_Q + (PFB_Q-1-biy);

    // LOCAL BUFFERS

    __shared__ float sm_input_buff_real[TPB+PFB_M-1];

    __shared__ float sm_input_buff_imag[TPB+PFB_M-1];

    __shared__ float sm_mac_real[TPB];

    __shared__ float sm_mac_imag[TPB];

    // INITIALIZE

    sm_mac_real[tix] = 0;

    sm_mac_imag[tix] = 0;

    __syncthreads();

    if (pdx < NUM_SAMPLES)

    {

        // STEP 1: global coalesced load from GM (shuffled data) to SM

        sm_input_buff_real[tix+PFB_M-1] = d_x_real[idx];

        sm_input_buff_imag[tix+PFB_M-1] = d_x_imag[idx];

        __syncthreads();

        // STEP 1.5: grab previous M-1 samples

        if (tix < PFB_M-1)

        {

            if (bix == 0)

            {

                sm_input_buff_real[tix] = 0;
```

```
                sm_input_buff_imag[tix] = 0;

            }

            else

            {

                sm_input_buff_real[tix] = d_x_real[idx-PFB_M+1];

                sm_input_buff_imag[tix] = d_x_imag[idx-PFB_M+1];

            }

        }

        __syncthreads();

        // STEP 2: MAC operation

        for (ii = 0; ii < PFB_M; ++ii)

        {

            sm_mac_real[tix] += cm_h[(PFB_M-1-ii)*PFB_Q+biy]*

            sm_input_buff_real[tix+PFB_M-1-ii];

            sm_mac_imag[tix] += cm_h[(PFB_M-1-ii)*PFB_Q+biy]*

            sm_input_buff_imag[tix+PFB_M-1-ii];

        }

        __syncthreads();

        // STEP 3: write back to GM

        d_fft_input[rdx].x = sm_mac_real[tix];

        d_fft_input[rdx].y = sm_mac_imag[tix];

    }

    else
```

```
        return;
}
```

A.2 TM Resampling

In this section, we provide CUDA code for a 1D texture memory (TM) kernel call used to perform the resampling option used in [64], and reported on in Chapter 5 of this thesis.

```
__global__ void gpu_TM (float *d_tm_fft_result)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    float DEL_F = ((float)ARB_P)*((float)NUM_INPUTS)/((float)NUM_OUTPUTS);
    float2 out[NUM_OUTPUTS];
    if (idx < NUM_OUTPUTS)
    {
        out[idx] = tex1D(texRef, 0.5f + DEL_F*((float)idx));
        d_tm_fft_result[idx] = out[idx].x;
    }
}
```

A.3 Polyphase Interpolator

In this section, we provide sample CUDA code for the polyphase interpolator used in Chapter 5.

```
__global__ void gpu_poly_interp (float *d_x_real, float *d_x_imag,
                                 float2 *d_poly_interp_cplx_result)
{
    // LOCAL INDEX
    int ii = 0;
    int tix = threadIdx.x;
    int tiy = threadIdx.y;
    int bix = blockIdx.x;
    int biy = blockIdx.y;
    int pdx = blockIdx.x*blockDim.x+threadIdx.x;
    int qdx = pdx+biy*ONE_FRAME;
    int odx = (blockIdx.x*INT_P*NEW_SAMPS_PER_BLOCK) + (tix*INT_P+tiy);
    int rdx = odx+biy*(ONE_FRAME*INT_P);
    // LOCAL BUFFERS
    __shared__ float sm_buff_real[NEW_SAMPS_PER_BLOCK+INT_M-1];
    __shared__ float sm_buff_imag[NEW_SAMPS_PER_BLOCK+INT_M-1];
    // LOAD DATA
    if (tiy == 0)
    {
        sm_buff_real[tix+INT_M-1] = d_x_real[qdx];
        sm_buff_imag[tix+INT_M-1] = d_x_imag[qdx];
    }
    if (tiy == 0 && tix < INT_M-1)
```

```
{
    if (bix == 0)
    {
        sm_buff_real[tix] = 0;
        sm_buff_imag[tix] = 0;
    }
    else
    {
        sm_buff_real[tix] = d_x_real[qdx-INT_M+1];
        sm_buff_imag[tix] = d_x_imag[qdx-INT_M+1];
    }
}
__syncthreads();
float2 sum;
sum.x = 0;
sum.y = 0;
// COMPUTE IP
for (ii = 0; ii < INT_M; ++ii)
{
    sum.x += cm_int_h[ii*INT_P+tiy]*sm_buff_real[tix+INT_M-1-ii];
    sum.y += cm_int_h[ii*INT_P+tiy]*sm_buff_imag[tix+INT_M-1-ii];
}
// STORE DATA
```

```
    d_poly_interp_cplx_result[rdx] = sum;
}
```

A.4 Data Shuffling

In this section, we provide sample CUDA code for data shuffling prior to polyphase filtering to improve global memory load efficiency. This is related to the developments of Chapter 4 in this thesis.

```
__global__ void gpu_shuffle (float *d_x_real, float *d_x_imag,
                             float *d_s_real, float *d_s_imag)
{
    int idx = blockIdx.x*blockDim.x+threadIdx.x;
    int row = idx % PFB_Q;
    int col = idx / PFB_Q;
    if (idx < NUM_INPUTS)
    {
        d_s_real[col+row*NUM_SAMPLES] = d_x_real[idx];
        d_s_imag[col+row*NUM_SAMPLES] = d_x_imag[idx];
    }
    else
        return;
}
```

A.5 Data Reshuffling

In this section, we provide sample CUDA code for reshuffling the data at the output of polyphase filtering for improved global memory store efficiency. This is related to the developments of Chapter 5.

```
__global__ void gpu_reshuffle (float2 *d_in, float2 *d_out)
{
    int idx = threadIdx.y*blockDim.x + threadIdx.x;
    int new_tiy = idx / INT_P;
    int new_tix = idx % INT_P;
    if (idx < NUM_INPUTS)
    {
        d_out[odx].x = d_in[new_tiy][new_tix];
        d_out[odx].y = d_in[new_tiy][new_tix];
    }
}
```

Bibliography

[1] E. Venosa, F. J. Harris, and F. A. N. Palmieri. *Software Radio: Sampling Rate Selection, Design and Synchronization (Analog Circuits and Signal Processing)*. Springer, 2011.

[2] NVIDIA Corporation. *CUDA C Programming Guide*, July 2013.

[3] H. Meyr, M. Moeneclaey, and S. A. Fechtel. *Digital Communication Receivers, Synchronization, Channel Estimation, and Signal Processing*. Wiley-Interscience, 1997.

[4] U. Mengali and A. N. D'Andrea. *Synchronization Techniques for Digital Receivers*. Springer, 1997.

[5] J. A. C. Bingham. Multicarrier modulation for data transmission: an idea whose time has come. *IEEE Communications Magazine*, 28(5):5–14, 1990.

[6] E. Gutierrez, J. A Lopez-Salcedo, and G. Seco-Granados. Systematic design of transmitter and receiver architectures for flexible filter bank multi-carrier signals. *EURASIP Journal on Advances in Signal Processing*, 2014(103):1–26, 2014.

[7] Y. Wu and W. Y. Zou. Orthogonal frequency division multiplexing: a multicarrier modulation scheme. *IEEE Transactions on Consumer Electronics*, 41(3):392–399, 1995.

[8] B. Farhang-Boroujeny. Ofdm versus filter bank multicarrier. *IEEE Signal Processing Magazine*, 28(3):92–112, 2011.

[9] A. Ling and L. B. Milstein. Comparison of multi-carrier modulation techniques. In *Proceedings of the Military Communications Conference*, pages 731–739, 2005.

[10] M. Renfors, P. Siohan, B. Farhang-Boroujeny, and F. Bader. Filter banks for next generation multicarrier wireless communications. *EURASIP Journal on Advances in Signal Processing*, 2010(314193):1–2, 2010.

[11] Y. Lin, S. Phoong, and P. P. Vaidyanathan. *Filter Bank Transceivers for OFDM and DMT Systems*. Cambridge University Press, 2010.

[12] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, 1993.

[13] F. J. Harris. *Multirate Signal Processing for Communication Systems*. Prentice Hall, 2004.

[14] F. J. Harris, C. Dick, and M. Rice. Digital receivers and transmitters using polyphase filter banks for wireless communications. *IEEE Transactions on Microwave Theory and Techniques*, 51(4):1395–1412, 2003.

[15] F. Harris and C. Dick. Polyphase channelizer performs sample rate change required for both matched filtering and channel frequency spacing. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 1283–1287, 2009.

[16] F. Harris, C. Dick, X. Chen, and E. Venosa. Wideband 160-channel polyphase filter bank cable tv channeliser. *IET Signal Processing*, 5(4):325–332, 2011.

[17] F. Harris, E. Venosa, and X. Chen. Multirate signal processing for software radio architectures. In R. Chellappa and S. Theodoridis, editors, *Academic Press Library in Signal Processing*, pages 643–673. Academic Press, 2013.

[18] X. Chen, F. Harris, E. Venosa, and B. D. Rao. Non-maximally decimated filter bank-based single-carrier receiver: a pathway to next-generation wideband communication. *EURASIP Journal on Advances in Signal Processing*, 2014(62):1–15, 2014.

[19] X. Chen, F. J. Harris, E. Venosa, and B. D. Rao. Non-maximally decimated analysis/synthesis filter banks: Applications in wideband digital filtering. *IEEE Transactions on Signal Processing*, 62(4):852–867, 2014.

[20] A. Franck. Arbitrary sample rate conversion with resampling filters optimized for combination with oversampling. In *Proceedings of the IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, pages 149–152, 2011.

[21] T. Hentschel. *Sample Rate Conversion in Software Configurable Radios*. Artech House, Inc., 2002.

[22] C. Dick and F. Harris. Options for arbitrary resamplers in FPGA-based modulators. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 777–781, 2004.

[23] T. A. Ramstad. Digital methods for conversion between arbitrary sampling frequencies. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 32(3):577–591, 1984.

[24] G. Evangelista. Design of digital systems for arbitrary sampling rate conversion. *Signal Processing*, 83(2):377–387, 2003.

[25] A. I. Russell and P. E. Beckmann. Efficient arbitrary sampling rate conversion with recursive calculation of coefficients. *IEEE Transactions on Signal Processing*, 50(4):854–865, 2002.

[26] F. Harris. Performance and design of Farrow filter used for arbitrary resampling. In *Proceedings of the International Conference on Digital Signal Processing*, pages 595–599, 1997.

[27] K. P. Prasad and P. Satyanarayana. Fast interpolation algorithm using FFT. *Electronic Letters*, 22(4):185–187, 1986.

[28] R. Lyons. *Understanding Digital Signal Processing*. Prentice Hall, 2010.

[29] D. Fraser. Interpolation by the FFT revisited-an experimental investigation. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(5):665–675, 1989.

[30] J. Schoukens, R. Pintelon, and H. Van Hamme. The interpolated fast Fourier transform: a comparative study. *IEEE Transactions on Instrumentation and Measurement*, 41(2):226–232, 1992.

[31] R. E. Crochiere and L. R. Rabiner. *Multirate Digital Signal Processing*. Prentice Hall, 1983.

[32] F. J. Harris and M. Rice. Multirate digital filters for symbol timing synchronization in software defined radios. *IEEE Journal on Selected Areas in Communications*, 19(12):2346–2357, 2001.

[33] C. Dick, B. Egg, and F. Harris. Architecture and simulation of timing synchronization circuits for the FPGA implementation of narrowband waveforms. In *Proceedings of the Software Defined Radio Technical Conference and Product Exposition*, 2006.

[34] M.-u.-R. Awan and P. Koch. Combined matched filter and arbitrary interpolator for symbol timing synchronization in SDR receivers. In *Proceedings of the International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 153–156, 2010.

[35] C. Dick, F. Harris, and M. Rice. Synchronization in software radios-carrier and timing recovery using fpgas. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 2000.

[36] F. Gardner. A BPSK/QPSK timing-error detector for sampled receivers. *IEEE Transactions on Communications*, 34(5):423–429, May 1986.

[37] M. Frerking. *Digital Signal Processing In Communications Systems*. Springer, 2010.

[38] F. M. Gardner. Interpolation in digital modems. I. fundamentals. *IEEE Transactions on Communications*, 41(3):501–507, 1993.

[39] L. Erup, F. M. Gardner, and R. A. Harris. Interpolation in digital modems. II. implementation and performance. *IEEE Transactions on Communications*, 41(6):998–1008, 1993.

[40] N. Wilt. *CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley Professional, 2013.

[41] S. C. Kim, W. L. Plishker, and S. S. Bhattacharyya. An efficient GPU implementation of an arbitrary resampling polyphase channelizer. In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, pages 231–238, Cagliari, Italy, October 2013.

[42] *NVIDIA CUDA C Programming Guide*, April 2012. Version 4.2.

[43] W. Tuttlebee. *Software Defined Radio: Enabling Technologies*. Wiley, 2002.

[44] S. C. Kim, W. L. Plishker, S. S. Bhattacharyya, and J. R. Cavallaro. GPU-based acceleration of symbol timing recovery. In *Proceedings of the Conference on Design and Architectures for Signal and Image Processing*, pages 1–8, Karlsruhe, Germany, October 2012.

[45] M. Awan, F. Harris, and P. Koch. Time and power optimizations in FPGA-based architectures for polyphase channelizers. In *ASILOMAR*, pages 914–918, 2011.

[46] Yongtao Wang, H. Mahmoodi, Lih-Yih Chiou, Hunsoo Choo, Jongsun Park, W. Jeong, and K. Roy. Hardware architecture and VLSI implementation of a low-power high-performance polyphase channelizer with applications to subband adaptive filtering. In *ICASSP*, pages V–97–100 vol.5, 2004.

[47] Y. Wang, H. Mahmoodi, L.-Y. Chiou, H. Choo, J. Park, W. Jeong, and K. Roy. Energy-efficient hardware architecture and VLSI implementation of a polyphase channelizer with applications to subband adaptive filtering. *Journal of Signal Processing Systems*, 58(2):125–137, 2010.

[48] J. Eberspächer, H.-J. Vögel, C. Bettstetter, and C. Hartmann. *GSM — Architecture, Protocols and Services*. Wiley, third edition, 2009.

[49] S. C. Kim and S. S. Bhattacharyya. Implementation of a high throughput low latency polyphase channelizer on GPUs. *EURASIP Journal on Advances in Signal Processing*, 2014(141):1–10, 2014.

[50] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, second edition, 2013. ISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online).

[51] M. Wu, Y. Sun, G. Wang, and J. R. Cavallaro. Implementation of a high throughput 3GPP turbo decoder on GPU. *Journal of Signal Processing Systems*, 65(2), 2011.

[52] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro. High throughput low latency LDPC decoding on GPU for SDR systems. In *Proceedings of the IEEE Global Conference on Signal and Information Processing*, pages 1258–1261, 2013.

[53] M. Wu, Y. Sun, S. Gupta, and J. R. Cavallaro. Implementation of a high throughput soft MIMO detector on GPU. *Journal of Signal Processing Systems*, 64(1):123–136, 2011.

[54] C. del Mundo, V. Adhinarayanan, and W.-c. Feng. Accelerating fast Fourier transform for wideband channelization. In *IEEE International Conference on Communications*, pages 4776–4780, 2013.

[55] P.-H. Horrein, C. Hennebert, and F. Pétrot. Integration of GPU computing in a software radio environment. *Journal of Signal Processing Systems*, 69(1):55–65, 2012.

[56] K. van der Veldt, R. van Nieuwpoort, A. L. Varbanescu, and C. Jesshope. A polyphase filter for GPUs and multi-core processors. In *Proceedings of the Workshop on High-Performance Computing for Astronomy*, pages 33–40, 2012.

[57] V. Adhinarayanan and W.-C. Feng. Wideband channelization for software-defined radio via mobile graphics processors. In *Proceedings of International Conference on Parallel and Distributed Systems*, pages 86–93, 2013.

[58] M. Awan, P. Koch, C. Dick, and F. Harris. FPGA implementation analysis of polyphase channelizer performing sample rate change required for both matched filtering and channel frequency spacing. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 414–418, 2010.

[59] M.-u.-R. Awan and P. Koch. Polyphase channelizer as bandpass filters in multi-standard software defined radios. In *Proceedings of the International Workshop on Cognitive Radio and Advanced Spectrum Management*, pages 59–63, 2009.

[60] M. Awan, M. M. Alam, P. Koch, and N. Behjou. Area efficient implementation of polyphase channelizer for multi-standard software radio receiver. In *Proceedings of the Karlsruhe Workshop on Software Radios*, pages 123–130, 2008.

[61] R. Mahesh, A. P. Vinod, E.M.-K. Lai, and A. Omondi. Filter bank channelizers for multi-standard software defined radio receivers. *Journal of Signal Processing Systems*, 62(2):157–171, 2011.

[62] H. Holma and A. Toskala, editors. *WCDMA for UMTS: HSPA Evolution and LTE*. Wiley, 2007.

[63] N. Whitehead and A. Fit-Florea. Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. NVIDIA Technical White Paper, 2011.

[64] S. C. Kim and S. S. Bhattacharyya. Implementation of a low complexity low latency arbitrary resampler on GPUs. In *Proceedings of the IEEE Dallas Circuits and Systems Conference*, pages 1–4, Dallas, Texas, October 2014.

[65] G. Wang, H. Shen, Y. Sun, J. R. Cavallaro, A. Vosoughi, and Y. Guo. Parallel interleaver design for a high throughput HSPA+/LTE multi-standard turbo decoder. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(5):1376–1389, 2014.

[66] M. Wu, B. Yin, G. Wang, C. Dick, J. R. Cavallaro, and C. Studer. Large-scale MIMO detection for 3GPP LTE: Algorithms and FPGA implementations. *IEEE Journal on Selected Topics in Signal Processing*, 8(5):916–929, 2014.

[67] M. Jorgovanovic, M. Pajic, G. Kvascev, and J. Popovic. FPGA design of arbitrary down-sampler. In *Proceedings of the International Conference on Microelectronics*, pages 391–394, 2008.

[68] B. H. Tietche, O. Romain, and B. Denby. A practical FPGA-based architecture for arbitrary-ratio sample rate conversion. *Journal of Signal Processing Systems*, 78(2):147–154, 2015.

[69] Y. Xu, H. Wang, and Z. Shen. Modified polyphase filter for arbitrary sampling rate conversion. In *Proceedings of the International Conference on Wireless Communications Networking and Mobile Computing*, pages 1–4, 2010.

[70] E. Dahlman, S. Parkvall, J. Skold, and P. Beming. *3G Evolution, HSPA and LTE for Mobile Broadband*. Academic Press, second edition, 2008.

[71] B. Farhang-Boroujeny. Filter bank spectrum sensing for cognitive radios. *IEEE Transactions on Signal Processing*, 56(5):1801–1811, 2008.

[72] B. Farhang-Boroujeny and R. Kempter. Multicarrier communication techniques for spectrum sensing and communication in cognitive radios. *IEEE Communications Magazine*, 46(4):80–85, 2008.

[73] M. Kim and J. Takada. Efficient multi-channel wideband spectrum sensing technique using filter bank. In *Proceedings of the International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1014–1018, 2009.

[74] S. C. Kim and S. S. Bhattacharyya. An efficient GPU implementation of a multirate resampler for multi-carrier systems. In *Proceedings of the IEEE Global Conference on Signal and Information Processing*, pages 1–5, Orlando, Florida, December 2015. (To appear).

[75] K. Adámek, J. Novotný, and W. Armour. The implementation of a real-time polyphase filter. *CoRR*, abs/1411.3656, 2014.

[76] E. Dahlman, S. Parkvall, and J. Skold. *4G: LTE/LTE–Advanced for Mobile Broadband*. Academic Press, 2011.

[77] K. Mueller and M. Muller. Timing recovery in digital synchronous data receivers. *IEEE Transactions on Communications*, 24(5):516–531, May 1976.

[78] B. Sklar. *Digital Communications: Fundamentals and Applications*. Prentice Hall, 2001.

[79] J. Vesma, M. Renfors, and J. Rinne. Comparison of efficient interpolation techniques for symbol timing recovery. In *Proceedings of the IEEE Global Telecommunications Conference*, pages 953–957, 1996.

[80] F. M. Gardner. *Phaselock Techniques*. Wiley-Interscience, third edition, 2005.