GPU Implementation of Data-Aided Equalizers

Jeffrey Thomas Ravert

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Michael Rice, Chair
Brian D. Jeffs
Karl F. Warnick

Department of Electrical and Computer Engineering

Brigham Young University

ABSTRACT

GPU Implementation of Data-Aided Equalizers

Jeffrey Thomas Ravert
Department of Electrical and Computer Engineering, BYU
Master of Science

Multipath is one of the dominant causes for link loss in aeronautical telemetry. Equalizers have been studied to combat multipath interference in aeronautical telemetry. Blind equalizers are currently being used with SOQPSK-TG. The Preamble Assisted Equalization (PAQ) project studied data-aided equalizers with SOQPSK-TG. PAQ compares, side-by-side, no equalization, blind equalization, and five data-aided equalization algorithms: ZF, MMSE, MMSE-initialized CMA, and frequency domain equalization. This thesis describes the GPU implementation of data-aided equalizer algorithms. Static lab tests, performed with channel and noise emulators, showed that the MMSE, ZF, and FDE1 show the best and most consistent performance.

Keywords: equalization, SOQPSK-TG, GPU, CUDA, aeronautical telemetry

ACKNOWLEDGMENTS

Acknowledgments and thanks to Dr. Rice for the countless hours of guidance he has offered in the past few years, and to my amazing wife Megan for putting me though my undergrad and staying home with our perfect little boy, Maxwell, while I work through my graduate degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LISTINGS

CHAPTER 1. INTRODUCTION

1.1 Multipath in Aeronautical Telemetry

Multipath interference is one of the dominant causes for link loss in aeronautical telemetry. Strong multipath interference occurs when the transmitted signal is received via multiple paths when a test article is in a low elevation angle scenario, as shown in Figure 1.1. Multipath propagation is modeled as linear, time-invariant system with a finite length impulse response. Equalizers have been studied to combat this form of multipath interference in aeronautical telemetry [1, 2]. There are two general classes of equalizers, blind and data-aided. Blind equalizers are adaptive filters whose coefficient update algorithm is based on the known statistical properties of the transmitted signal, but have no knowledge of the specific data or multipath channel is assumed. Data-aided equalizers are also filters whose impulse responses are calculated from the propagation conditions. One method of obtaining an estimate of this knowledge is to periodically insert a known bit sequence called a "pilot" into the data stream. The receiver compares the received signal corresponding to the pilot with a locally stored copy to estimate parameters such as the multipath channel, frequency offset, phase offset, and noise variance.

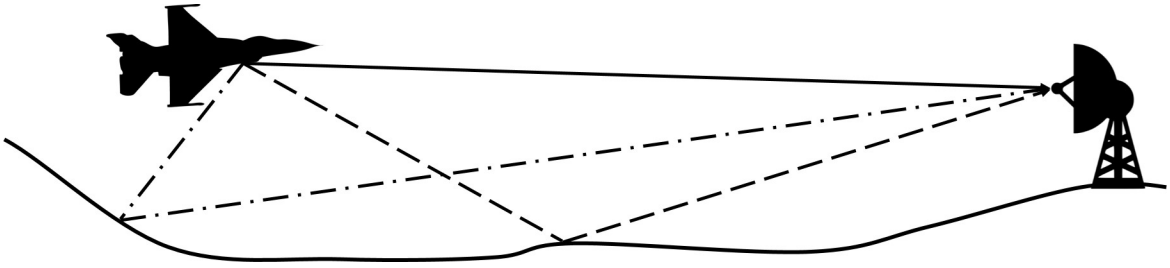


Figure 1.1: Multipath interference can occur when a signal is received from multiple paths.

1.2 Problem Statement

The real-time signal processing for a digital communications system with data-aided equalizers is computationally heavy. Digital communication algorithms implemented in a high performance Central Processing Unit (CPU) cannot meet the real-time requirement. Graphic Processing Units (GPUs) can be used to perform real-time processing because of their massively parallel architecture.

This thesis studies how signal processing can be reformulated to run quickly and efficiently in GPUs. Optimized libraries harness GPU resources to make signal processing implementation relatively easy and extremely fast. If algorithms can be reformulated for batch processing and use matrix/vector multiplication or Fast Fourier Transform libraries, GPUs can provide vast speed ups.

1.3 Organization

Chapter 2 describes the Preamble Assisted Equalization (PAQ) system and introduces the digital signal processing algorithms. Chapter 3 provides an overview of signal processing in GPUs. Chapter 4 describes how the five equalizers are implemented in GPUs. The thesis concludes with the summary and conclusions in Chapter 5.

CHAPTER 2. PAQ PROJECT

Data-aided equalization in aeronautical telemetry has been studied and tested by the Preamble Assisted Equalization (PAQ) project [3]. Under PAQ, a system was built that compared five data-aided equalizers to blind equalization and no equalization. Laboratory tests were performed using a static RF multipath channel emulator and a noise source to produce bit error rate (BER) curves. The five data-aided equalizers studied are:

- zero-forcing (ZF) equalizer,
- minimum mean-square error (MMSE) equalizer,
- MMSE-initialized constant modulus algorithm (CMA) equalizer,
- frequency domain equalizer one (FDE1), and
- frequency domain equalizer two (FDE2).

Bit error rate statistics were used as the figure or merit for the equalization algorithms.

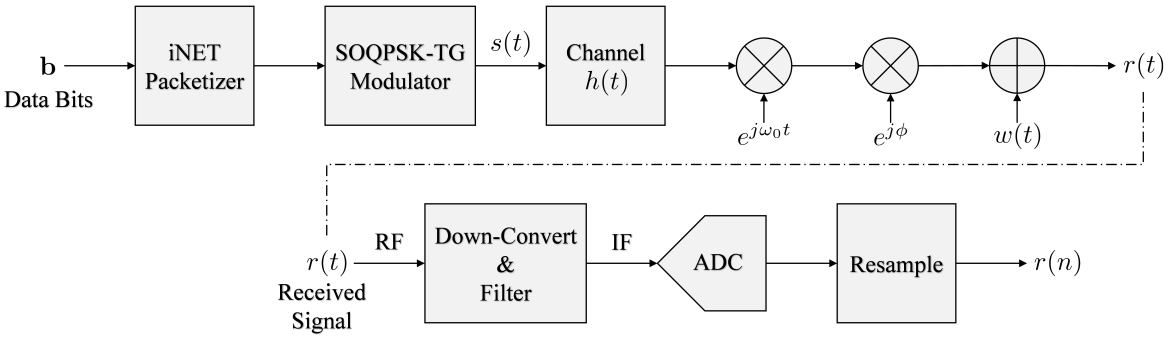


Figure 2.1: The received signal has multipath interference, frequency offset, phase offset, and additive white Gaussian noise. The received signal is down-converted, filtered, sampled, and resampled to produce the sample sequence $r(n)$.

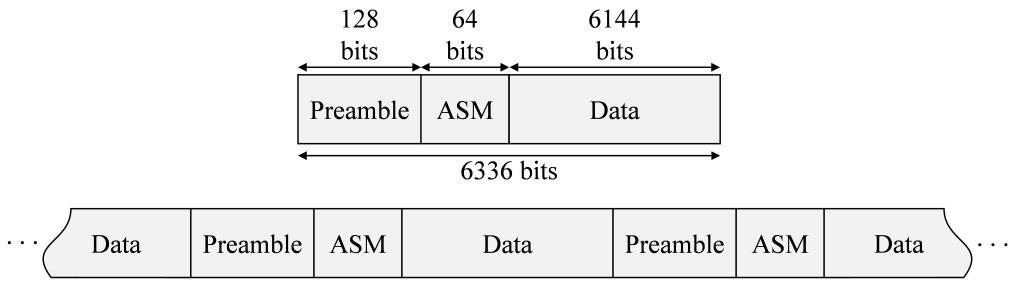


Figure 2.2: A diagram showing each PAQ packet comprises a preamble, ASM, and a data field.

2.1 System Overview

The system is summarized by the block diagram in Figure 2.1. To enable data-aided equalization, the PAQ bit stream has a packetized structure shown in Figure 2.2. Each packet comprises a preamble (defined in the iNET standard [4]), the attached sync marker (ASM), and a 6144-bit data field. The preamble and ASM bits form a known sequence of bits, together called the pilot bits, that are periodically inserted every 6144 bits. The iNET preamble comprises eight repetitions of the 16-bit sequence $\mathrm{CD98}_{\mathrm{hex}}$ and the ASM field is

$$034776\mathrm{C}7272895\mathrm{B}0_{\mathrm{hex}}. \tag{2.1}$$

The data payload is a known length-$(2^{11}-1)$ PN sequence. Each packet contains 128 preamble bits, 64 ASM bits and 6,144 data bits making each iNET packet 6,336 bits. The data bit rate is 10 Mbits/s. After preamble and ASM insertion, the bit rate presented to the modulator is 10.3125 Mbits/s.

After modulation, the transmitted signal experiences multipath interference modeled as an LTI system with the channel impulse response $h(t)$. The transmitted signal also experiences a frequency offset ω_0, a phase offset ϕ and additive white Gaussian noise $w(t)$. The received signal is down-converted, filtered, sampled at $93^1/_3$ Msamples/second by the ADC, and down-converted to baseband and resampled by $^{99}/_{448}$ using a polyphase filterbank based on the principles outlined in [5, chap. (9)]. The result is $r(n)$, a sampled version of the complex-valued lowpass equivalent waveform at a sample rate of 20.625 Msamples/second or 2 samples/bit.

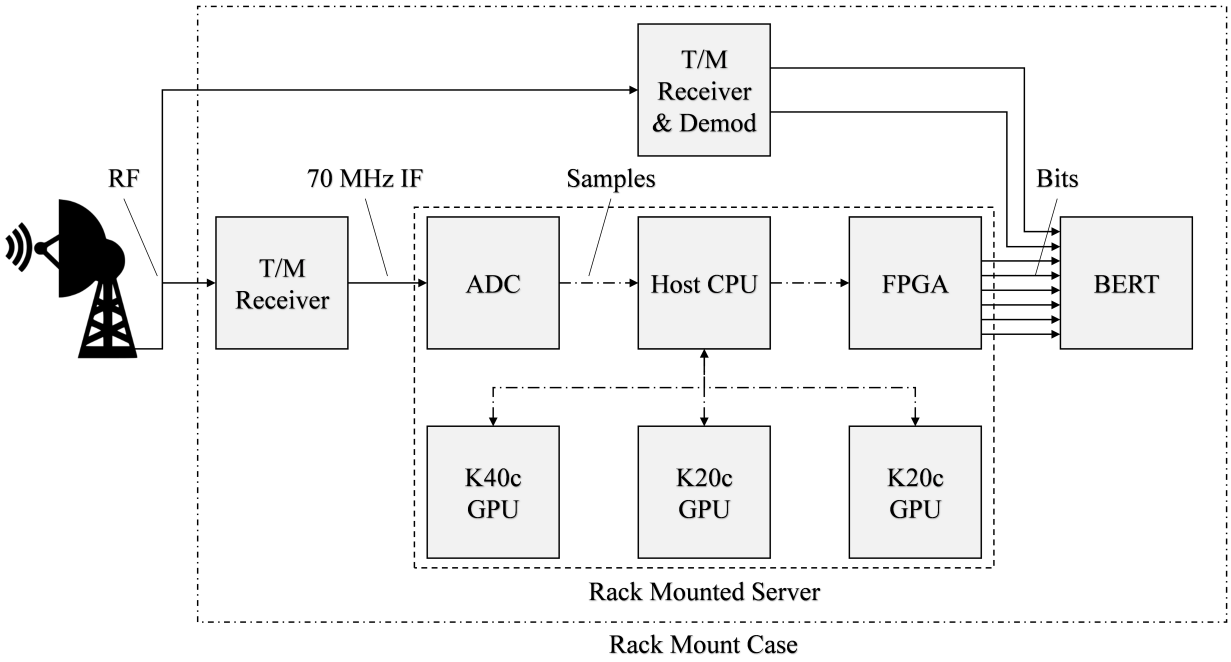


Figure 2.3: A block diagram of the physical PAQ hardware. The components inside the rack mounted server are in the dashed box. All the components in the dashed and dotted box are housed in a rack mounted case.

2.2 Hardware Overview

A block diagram of PAQ physical system is shown in Figure 2.3. A picture of the physical components is shown in Figure 2.4. The major components, and their functions are summarized as follows:

- The **T/M receiver** down-converts the received signal from L- or C-band RF to 70 MHz IF. The IF filter plays the role of an anti-aliasing filter.
- The **ADC** produces 14-bit samples of the real-valued bandpass IF signal. The sample rate is $93^1/_3$ Msamples/s. The samples are transferred to the host CPU via the PCIe bus.
- The **host CPU** initiates memory transfers between itself and the ADC, GPUs and FPGA via the PCIe bus. The host CPU also launches the digital signal processing algorithms on the GPUs.
- The three **GPUs** are where the detection, estimation, equalization and demodulation resides.

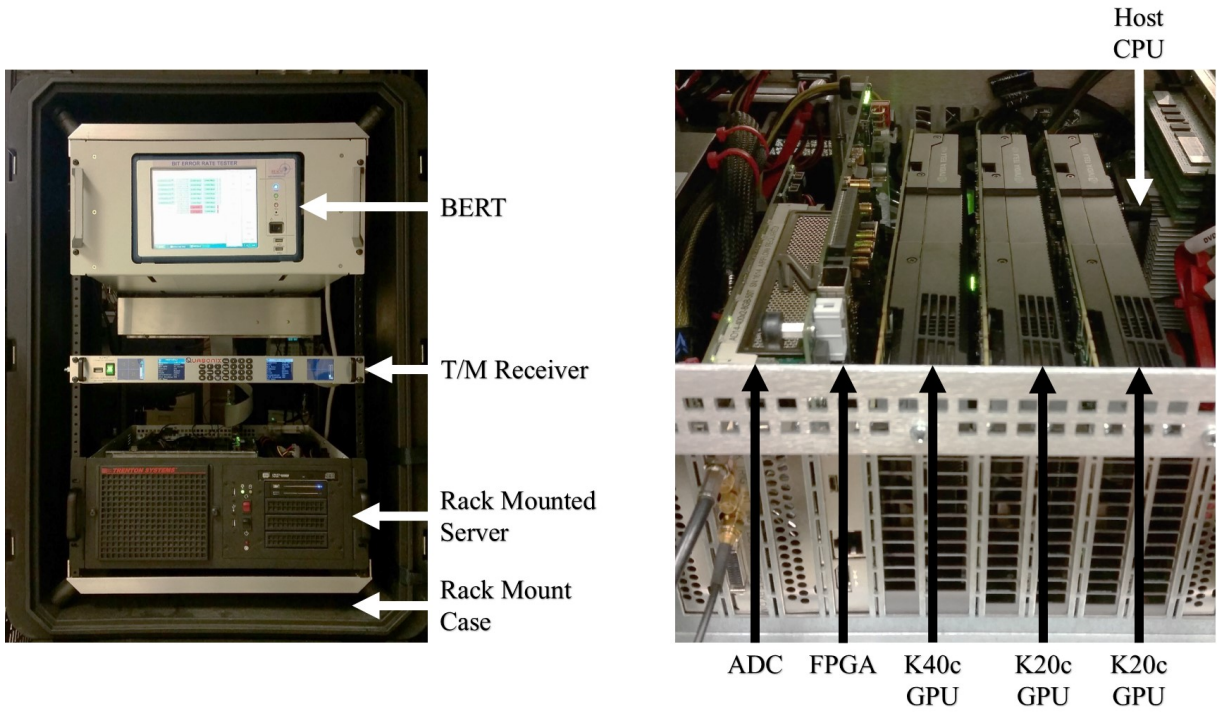


Figure 2.4: A picture of the physical PAQ hardware refrencing blocks from Figure 2.3. Right: Components in the dashed and dotted box. Left: Components in the dashed box. Note that the T/M Receiver is not pictured.

- The bit error rate tester (**BERT**) counts the errors in each bit stream by comparing the streams to the transmitted PN sequence.
- The **FPGA** is the interface between the host CPU and the BERT. After the GPUs produce bit decisions, the host CPU transfers the decisions from the GPUs to the FPGA via the PCIe bus. The FPGA then clocks the bits out to the BERT for BER testing.
- The **T/M Receiver & Demodulator** demodulates the RF signal outputting two bit streams for blind equalization and no equalization for BER comparison.
- The **rack mounted server** is a high powered computer that houses an ADC, a FPGA and three GPUs slotted into a 32 pin PCIe bus.

2.3 Digital Signal Processing

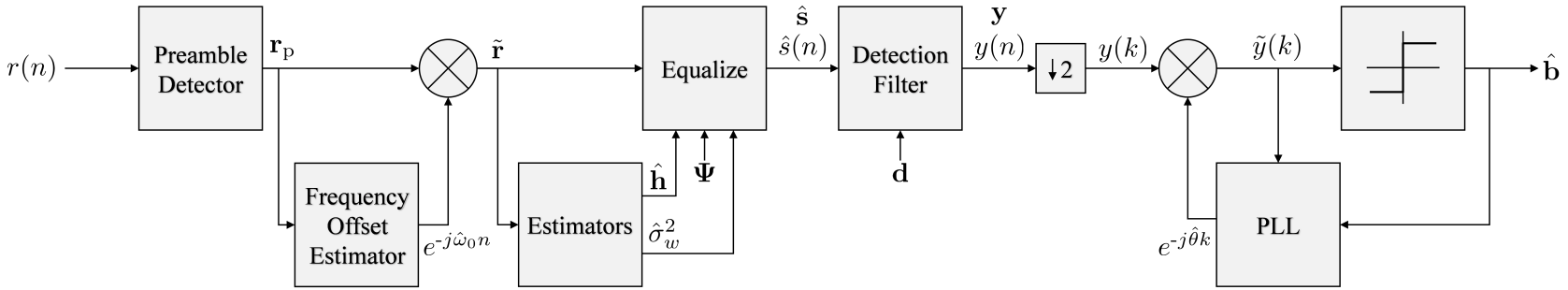
A high-level digital signal processing flow and notation is shown in Figure 2.5. The sequence $r(n)$ represents a continuous stream of samples. Because the frequency offset, channel, and noise variance are estimated using the preamble and ASM, the first step is to find the samples corresponding to the preamble in the received sample sequence $r(n)$. The preamble detector identifies the sample indices in the sequence $r(n)$ corresponding to the starting position of each occurrence of the waveform samples corresponding to the preamble. To simplify the notation used to describe the signal processing algorithms, we represent the output of the preamble detector by the vector $\mathbf{r}_{\mathrm{p}}$, a sequence of L_{pkt} samples starting with the waveform samples corresponding to the preamble and ASM bits. In this way the signal processing algorithms are described on a packet-by-packet basis.

Starting with the block diagram in Figure 2.5, the preamble samples are used first to estimate the frequency offset. The estimated frequency offset $\hat{\omega}_0$ rads/sample is then used to "de-rotate" the vector of samples $\mathbf{r}_{\mathrm{p}}$ to produce a vector denoted $\tilde{\mathbf{r}}$. The de-rotated preamble and ASM samples in the vector $\tilde{\mathbf{r}}$ are used to estimate the channel $\hat{\mathbf{h}}$ and noise variance $\hat{\sigma}_w^2$ as shown.

The estimates are used to compute the equalizer filter coefficients as illustrated in Figure 2.6. The figure shows five independent branches, each branch computing an equalizer filter. On the top three branches, lower case boldface variables $\mathbf{c}$, with a subscript, represent the impulse responses of the FIR equalizer filters. On the lower two branches, the upper case boldface $\mathbf{C}$ with a subscript represent the FFT-domain transfer function of the equalizers. In all five cases, the equalizer and a detection filter (described below) are applied to $\tilde{\mathbf{r}}$. The result processed by a symbol-by-symbol OQPSK detector to produce bit decisions for each equalizer.

The GPUs in Figure 2.3 and 2.4 perform all the digital signal processing in parallel. To introduce as much parallelism as possible, the received samples are processed in a batch comprising 39,321,600 samples. At 20.625 Msamples/second, each batch of 39,321,600 samples represents 1907 milliseconds of data. Each batch has at most 3104 12,672-sample iNET packets.[1] The GPU processes 3104 packets in parallel by leveraging batched processing. To meet the real-time requirement, all processing must be completed in less than 1907 ms.

[1] Because the batch length (39,321,600 samples) is not a multiple of the packet length (12,672), each batch comprises 3103 or 3104 packets.



$$\mathbf{r}_\mathrm{p} = \begin{bmatrix} r(i) \\ \vdots \\ r(i+L_\mathrm{pkt}-1) \end{bmatrix} \quad \tilde{\mathbf{r}} = \begin{bmatrix} r_\mathrm{p}(0)e^{-j\hat{\omega}_0(0)} \\ \vdots \\ r_\mathrm{p}(L_\mathrm{pkt}-1)e^{-j\hat{\omega}_0(L_\mathrm{pkt}-1)} \end{bmatrix} \quad \hat{\mathbf{s}} = \begin{bmatrix} \hat{s}(0) \\ \vdots \\ \hat{s}(L_\mathrm{pkt}-1) \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} y(0) \\ \vdots \\ y(L_\mathrm{pkt}-1) \end{bmatrix} \quad \hat{\mathbf{b}} = \begin{bmatrix} \hat{b}(0) \\ \vdots \\ \hat{b}(6{,}336-1) \end{bmatrix}$$

$$\hat{\mathbf{h}} = \begin{bmatrix} \hat{h}(0) \\ \vdots \\ \hat{h}(N_1+N+2) \end{bmatrix} \quad \mathbf{\Psi} = \begin{bmatrix} \Psi(0) \\ \vdots \\ \Psi(32{,}768-1) \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} d(0) \\ \vdots \\ d(L_\mathrm{df}-1) \end{bmatrix}$$

Figure 2.5: A block diagram of the digital signal processing flow and notation in PAQ.

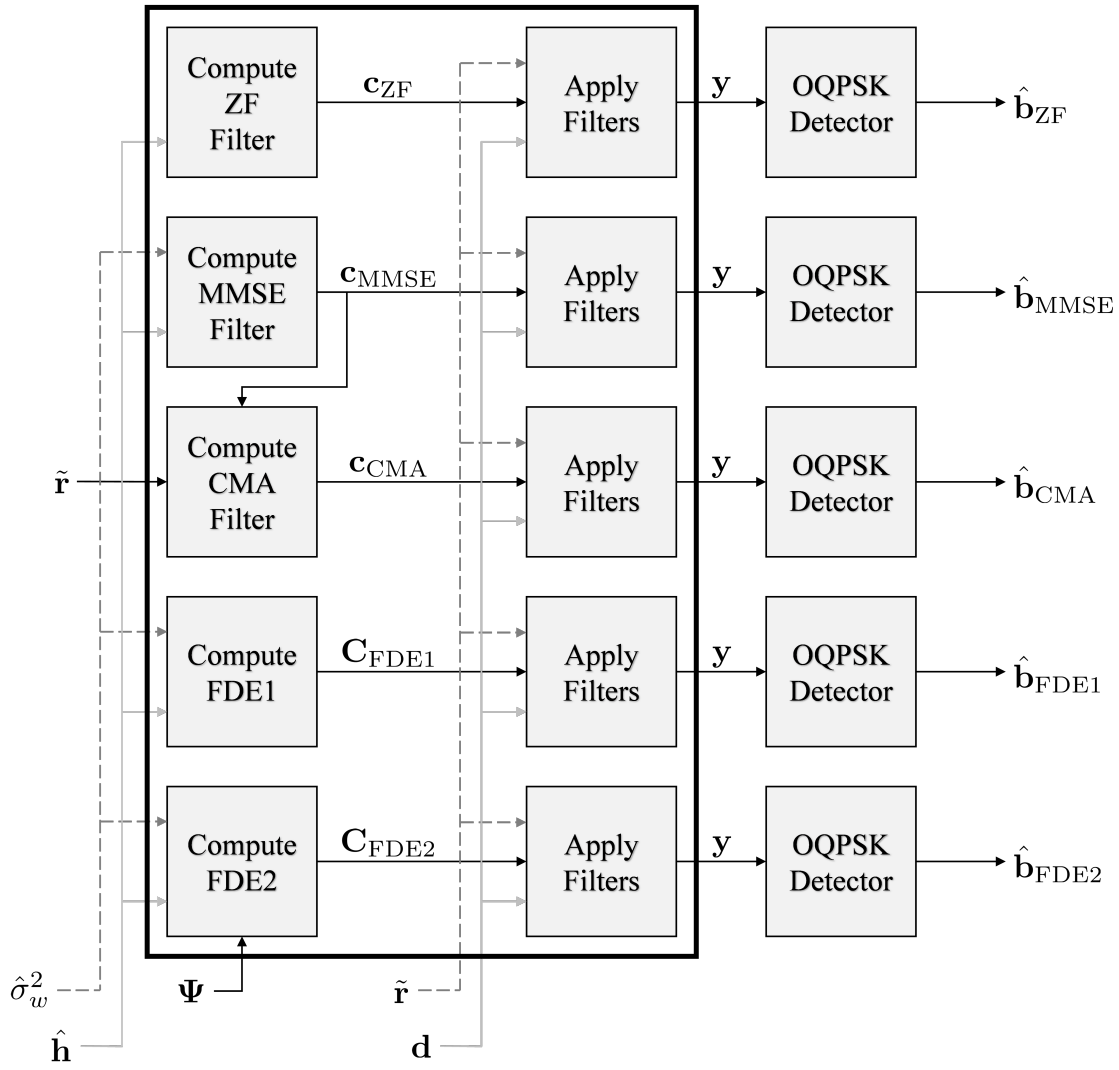


Figure 2.6: A block diagram of the computation and application of the equalizer and detection filters. The bold box emphasizes the focus of this thesis.

This thesis illustrates how the five PAQ data-aided equalizers were computed and applied in GPUs. The bold box in Figure 2.6 emphasizes processing blocks on which this thesis focuses. Even though the GPUs process 3104 packets in parallel, the signal processing algorithms are described on a packet-by-packet basis.

2.3.1 Preamble Detection

To compute the impulse responses or transfer functions of the five data-aided equalizers, an estimate of the channel and noise variance must be available. The required estimates are derived

from the received waveform samples corresponding to the preamble and ASM bits. Consequently, the location of the waveform samples corresponding to the preamble and ASM bits must be found. The preamble detector identifies the sample indices in the sequence $r(n)$ corresponding to the starting position of each occurrence of the waveform samples corresponding to the preamble. The preamble detector computes the function $L(n)$ for each sample in the batch. Peaks in $L(n)$ identify the starting indices of the waveform samples corresponding to each occurrence of the preamble bits. The function $L(n)$ is given by [6]

$$L(n) = \sum_{m=0}^{7} \left[I^2(n,m) + Q^2(n,m) \right], \tag{2.2}$$

where

$$\begin{aligned} I(n,m) \approx & \sum_{\ell \in \mathscr{L}_1} r_R(\ell + 32m + n) - \sum_{\ell \in \mathscr{L}_2} r_R(\ell + 32m + n) + \sum_{\ell \in \mathscr{L}_3} r_I(\ell + 32m + n) \\ & - \sum_{\ell \in \mathscr{L}_4} r_I(\ell + 32m + n) + 0.7071 \left[\sum_{\ell \in \mathscr{L}_5} r_R(\ell + 32m + n) - \sum_{\ell \in \mathscr{L}_6} r_R(\ell + 32m + n) \right. \\ & \left. + \sum_{\ell \in \mathscr{L}_7} r_I(\ell + 32m + n) - \sum_{\ell \in \mathscr{L}_8} r_I(\ell + 32m + n) \right], \end{aligned} \tag{2.3}$$

and

$$\begin{aligned} Q(n,m) \approx & \sum_{\ell \in \mathscr{L}_1} r_I(\ell + 32m + n) - \sum_{\ell \in \mathscr{L}_2} r_I(\ell + 32m + n) \\ & - \sum_{\ell \in \mathscr{L}_3} r_R(\ell + 32m + n) + \sum_{\ell \in \mathscr{L}_4} r_R(\ell + 32m + n) \\ & + 0.7071 \left[\sum_{\ell \in \mathscr{L}_5} r_I(\ell + 32m + n) - \sum_{\ell \in \mathscr{L}_6} r_I(\ell + 32m + n) \right. \\ & \left. - \sum_{\ell \in \mathscr{L}_7} r_R(\ell + 32m + n) + \sum_{\ell \in \mathscr{L}_8} r_R(\ell + 32m + n) \right], \end{aligned} \tag{2.4}$$

with

$$\begin{aligned}
\mathscr{L}_1 &= \{0,8,16,24\} \\
\mathscr{L}_2 &= \{4,20\} \\
\mathscr{L}_3 &= \{2,10,14,22\} \\
\mathscr{L}_4 &= \{6,18,26,30\} \\
\mathscr{L}_5 &= \{1,7,9,15,17,23,25,31\} \\
\mathscr{L}_6 &= \{3,5,11,12,13,19,21,27,28,29\} \\
\mathscr{L}_7 &= \{1,3,9,11,12,13,15,21,23\} \\
\mathscr{L}_8 &= \{5,7,17,19,25,27,28,29,31\}.
\end{aligned} \tag{2.5}$$

The index of a peak in $L(n)$ indicates the start of a preamble. Suppose $L(i)$ is a peak (i.e., i is the index of the peak). Then the vector $\mathbf{r}_\mathrm{p}$ in Figure 2.5 is

$$\mathbf{r}_\mathrm{p} = \begin{bmatrix} r(i) \\ \vdots \\ r(i+L_\mathrm{pkt}-1) \end{bmatrix} = \begin{bmatrix} r_\mathrm{p}(0) \\ \vdots \\ r_\mathrm{p}(L_\mathrm{pkt}-1) \end{bmatrix}. \tag{2.6}$$

The first $L_\mathrm{p} = 256$ samples of $\mathbf{r}_\mathrm{p}$ correspond to the preamble bits and the following $L_\mathrm{ASM} = 128$ samples of $\mathbf{r}_\mathrm{p}$ correspond to the ASM bits.

2.3.2 Frequency Offset Compensation

The preamble sequence comprises eight copies of the bit sequence $\mathrm{CD98}_\mathrm{hex}$. Consequently, the waveform samples $r_\mathrm{p}(0),\ldots,r_\mathrm{p}(L_\mathrm{p}-1)$ comprise eight copies of $L_q = 32$ SOQPSK-TG waveform samples corresponding to $\mathrm{CD98}_\mathrm{hex}$.[2] The frequency offset estimator shown in Figure 2.5 is the estimator taken from [7, eq. (24)]. With the notation adjusted slightly, the frequency offset estimate is

$$\hat{\omega}_0 = \frac{1}{L_q} \arg\left\{ \sum_{n=i+2L_q}^{i+7L_q-1} r_\mathrm{p}(n) r_\mathrm{p}^*(n-L_q) \right\} \quad \text{for } i = 1,2,3,4,5. \tag{2.7}$$

[2] This statement is only approximately true. Because of the memory in SOQPSK-TG, the first block of L_q samples is a function of both the bit sequence $\mathrm{CD98}_\mathrm{hex}$ and the seven unknown bits preceding the first occurrence of $\mathrm{CD98}_\mathrm{hex}$.

The frequency offset is estimated for every packet or each vector of samples $\mathbf{r}_\mathrm{p}$ in the batch. Frequency offset compensation is performed by de-rotating the received samples by $-\hat{\omega}_0$:

$$\tilde{r}(n) = r_\mathrm{p}(n)e^{-j\hat{\omega}_0 n}. \tag{2.8}$$

Equations (2.7) and (2.8) are easily implemented into GPUs.

2.3.3 Channel Estimation

Let the SOQPSK-TG samples corresponding to the preamble and ASM bits be

$$\mathbf{p} = \begin{bmatrix} p(0) \\ p(1) \\ \vdots \\ p(L_\mathrm{P} + L_\mathrm{ASM} - 1) \end{bmatrix}. \tag{2.9}$$

The multipath channel is defined by the impulse response

$$\hat{\mathbf{h}} = \begin{bmatrix} \hat{h}(-N_1) \\ \vdots \\ \hat{h}(0) \\ \vdots \\ \hat{h}(N_2) \end{bmatrix}. \tag{2.10}$$

Note that at 2 samples/bit, the complex-valued lowpass equivalent channel impulse response is assumed to have a non-causal component comprising N_1 samples and a causal component comprising N_2 samples. Figure 2.7 shows the full discrete-time $L_h = N_1 + N_2 + 1$ sample channel.
The ML estimate is [2, eq. 8]

$$\hat{\mathbf{h}} = \underbrace{\left(\mathbf{X}^\dagger \mathbf{X}\right)^{-1} \mathbf{X}^\dagger}_{\mathbf{X}_\mathrm{lpi}} \tilde{\mathbf{r}}_x, \tag{2.11}$$

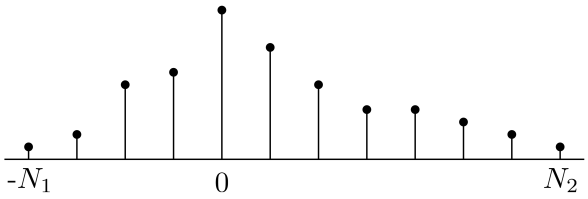


Figure 2.7: An illustration of the discrete-time channel of length $N_1 + N_2 + 1$ with a non-causal component comprising N_1 samples and a causal component comprising N_2 samples.

where

$$\mathbf{X} = \begin{bmatrix} p(N_2) & & & \\ \vdots & p(N_2) & & \\ p(L_\mathrm{p} + L_\mathrm{ASM} - N_1) & \vdots & \ddots & \\ & p(L_\mathrm{p} + L_\mathrm{ASM} - N_1) & & p(N_2) \\ & & & \vdots \\ & & & p(L_\mathrm{p} + L_\mathrm{ASM} - N_1) \end{bmatrix}, \tag{2.12}$$

is the $(L_\mathrm{p} + L_\mathrm{ASM} - N_1 - N_2) \times (N_1 + N_2 + 1)$ convolution matrix formed from the SOQPSK-TG waveform samples corresponding to the preamble and ASM bits and $\tilde{\mathbf{r}}_x$ is the vector of de-rotated received waveform samples corresponding to the "middle" portion of the preamble and ASM bits:

$$\tilde{\mathbf{r}}_x = \begin{bmatrix} \tilde{r}(N_2) \\ \tilde{r}(N_2 + 1) \\ \vdots \\ \tilde{r}(L_\mathrm{P} + L_\mathrm{ASM} - N_1) \end{bmatrix}. \tag{2.13}$$

The $(N_1 + N_2 + 1) \times (L_\mathrm{p} + L_\mathrm{ASM} - N_1 - N_2)$ matrix $\mathbf{X}_\mathrm{lpi}$ is the left pseudo-inverse of $\mathbf{X}$. Note that $\mathbf{X}_\mathrm{lpi}$ is independent of the data and therefore may be computed once and stored. The matrix vector multiplication $\mathbf{X}_\mathrm{lpi}\tilde{\mathbf{r}}_x$ is implemented simply and efficiently in GPUs.

2.3.4 Noise Variance Estimation

The noise variance estimator is [2, eq. 9]

$$\hat{\sigma}_w^2 = \frac{1}{2\rho} \left| \tilde{\mathbf{r}}_x - \mathbf{X}\hat{\mathbf{h}} \right|^2, \tag{2.14}$$

where

$$\rho = \text{Trace} \left\{ \mathbf{I} - \mathbf{X} \left(\mathbf{X}^\dagger \mathbf{X} \right)^{-1} \mathbf{X}^\dagger \right\}, \tag{2.15}$$

where $\tilde{\mathbf{r}}_x$ is given by Equation (2.13) and $\mathbf{X}$ is given by Equation (2.12). Equation (2.14) is easily implemented into GPUs.

2.3.5 Equalizers

Zero-Forcing Equalizer

The ZF equalizer is an FIR filter defined by the $L_{\text{eq}} = L_1 + L_2 + 1$ coefficients

$$\mathbf{c}_{\text{ZF}} = \begin{bmatrix} c_{\text{ZF}}(-L_1) \\ \vdots \\ c_{\text{ZF}}(0) \\ \vdots \\ c_{\text{ZF}}(L_2) \end{bmatrix}. \tag{2.16}$$

The filter coefficients are the solution to [3]

$$\mathbf{R}_{\hat{h}} \mathbf{c}_{\text{ZF}} = \hat{\mathbf{g}}, \tag{2.17}$$

where

$$\mathbf{R}_{\hat{h}} = \begin{bmatrix} r_{\hat{h}}(0) & r_{\hat{h}}^*(1) & \cdots & r_{\hat{h}}^*(L_{eq}-1) \\ r_{\hat{h}}(1) & r_{\hat{h}}(0) & \cdots & r_{\hat{h}}^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_{\hat{h}}(L_{eq}-1) & r_{\hat{h}}(L_{eq}-2) & \cdots & r_{\hat{h}}(0) \end{bmatrix}, \tag{2.18}$$

$$\hat{\mathbf{g}} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix}, \tag{2.19}$$

and

$$r_{\hat{h}}(k) = \sum_{n=-N_1}^{N_2} \hat{h}(n)\hat{h}^*(n-k). \tag{2.20}$$

MMSE Equalizer

The MMSE equalizer is an FIR filter defined by the $L_{\text{eq}} = L_1 + L_2 + 1$ coefficients

$$\mathbf{c}_{\text{MMSE}} = \begin{bmatrix} c_{\text{MMSE}}(-L_1) \\ \vdots \\ c_{\text{MMSE}}(0) \\ \vdots \\ c_{\text{MMSE}}(L_2) \end{bmatrix}. \tag{2.21}$$

The filter coefficients are the solution to [3]

$$\mathbf{R}\mathbf{c}_{\text{MMSE}} = \hat{\mathbf{g}}, \tag{2.22}$$

where

$$\mathbf{R} = \mathbf{R}_{\hat{h}} + \hat{\sigma}_w^2 \mathbf{I}, \tag{2.23}$$

$\mathbf{R}_{\hat{h}}$ is given by (2.18), $\hat{\sigma}_w^2$ is given by (2.14), and $\hat{\mathbf{g}}$ is given by (2.19).

Constant Modulus Algorithm Equalizer

The CMA equalizer is an adaptive FIR filter where the $L_{\text{eq}} = L_1 + L_2 + 1$ coefficients at the b-th iteration are

$$\mathbf{c}_{\text{CMA}}^{(b)} = \begin{bmatrix} c_{\text{CMA}}^{(b)}(-L_1) \\ \vdots \\ c_{\text{CMA}}^{(b)}(0) \\ \vdots \\ c_{\text{CMA}}^{(b)}(L_2) \end{bmatrix}. \tag{2.24}$$

The equalizer output at the b-th iteration is

$$\hat{\mathbf{s}}^{(b)} = \mathbf{c}_{\text{CMA}}^{(b)} * \tilde{\mathbf{r}}. \tag{2.25}$$

Note that in this implementation the CMA filter coefficients are constant for the duration of a packet [2]. The filter coefficients are updated on a packet-by-packet basis using a steepest descent algorithm as follows:

$$\mathbf{c}_{\text{CMA}}^{(b+1)} = \mathbf{c}_{\text{CMA}}^{(b)} - \mu \nabla J, \tag{2.26}$$

where

$$\nabla J = \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \left[\hat{s}^{(b)}(n) \left(\hat{s}^{(b)}(n) \right)^* - 1 \right] \hat{s}^{(b)}(n) \tilde{\mathbf{r}}^*(n). \tag{2.27}$$

In Equation (2.27), $\hat{s}^{(b)}(n)$ is the n-th element of the vector $\hat{\mathbf{s}}^{(b)}$ and

$$\tilde{\mathbf{r}}^*(n) = \begin{bmatrix} \tilde{r}^*(n+L_1) \\ \vdots \\ \tilde{r}^*(n) \\ \vdots \\ \tilde{r}^*(n-L_2) \end{bmatrix}. \tag{2.28}$$

The CMA equalizer filter coefficients are initialized by the MMSE equalizer filter coefficients

$$\mathbf{c}_{\text{CMA}}^{(0)} = \mathbf{c}_{\text{MMSE}}. \tag{2.29}$$

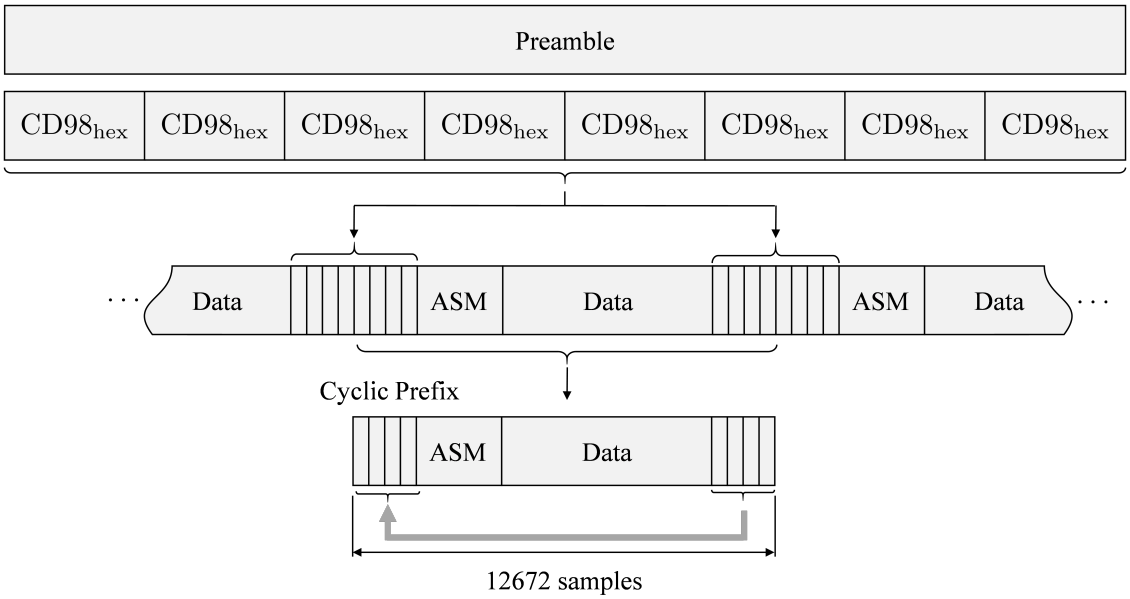


Figure 2.8: A diagram showing how the iNET packet is used as a cyclic prefix.

Frequency Domain Equalizer One

Frequency-domain equalization leverages the efficiency of the FFT algorithm to perform equalization filtering in the FFT domain. The difference between frequency-domain equalization and applying the previous three equalizer filters in the FFT domain is that the frequency-domain equalizer is computed directly in the FFT domain. To enable this, some provision must be made for the fact that point-by-point multiplication in the FFT domain corresponds to circular convolution in the time domain. This provision is most often in the form of a cyclic prefix prepended to the data packet [8–11]. Even though the PAQ format does not include any special provision for frequency-domain equalization such as a cyclic prefix, frequency-domain equalization is still possible using the ideas described by Coon et al [12]. Because of the repetitive nature of the preamble sequence, the second half of the preamble bits at the beginning of the packet are the same the first half of the preamble bits following the packet. Consequently, the second half of the preamble bits at the beginning of the packet form a cyclic prefix for the block comprising the ASM, the data, and the first half of the preamble following the packet as illustrated in Figure 2.8.

The FFT domain transfer function of FDE1 is [13, eq. (11)]

$$C_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{\text{FFT}}} \text{ for } k = 0, 1, \cdots, N_{\text{FFT}} - 1, \tag{2.30}$$

where $N_{\text{FFT}} = 2^u = 16{,}384$, where $u = \lceil \log_2 (L_{\text{pkt}}) \rceil = 14$, where $\lceil x \rceil$ means the smallest integer greater than or equal to x. In Equation (2.30), $\hat{H}(e^{j\omega_k})$ is the k-th element of the length-N_{FFT} FFT of $\hat{\mathbf{h}}$ and $\hat{\sigma}_w^2$ is given by (2.14). FDE1 is the MMSE equalizer formulated in the frequency domain, where power spectral density of SOQPSK-TG is a constant.

Frequency Domain Equalizer Two

The FFT domain transfer function of FDE1 is [13, eq. (12)]

$$C_{\text{FDE2}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{\text{FFT}}} \text{ for } k = 0, 1, \cdots, N_{\text{FFT}} - 1, \tag{2.31}$$

where $N_{\text{FFT}} = 2^u = 16{,}384$, where $u = \lceil \log_2 (L_{\text{pkt}}) \rceil = 14$, where $\lceil x \rceil$ means the smallest integer greater than or equal to x. In Equation (2.31), $\hat{H}(e^{j\omega_k})$ is the k-th element of the length-N_{FFT} FFT of $\hat{\mathbf{h}}$ and $\hat{\sigma}_w^2$ is given by (2.14). Like FDE1, FDE2 is the MMSE equalizer formulated in the frequency domain. The difference is FDE2 uses an estimate of the true power spectral density of SOQPSK-TG. The SOQPSK-TG power spectral density $\Psi(e^{j\omega_k})$ is illustrated in Figure 2.9. $\Psi(e^{j\omega_k})$ was estimated using Welch's method of periodogram averaging based on length-N_{FFT} FFTs of SOQPSK-TG sampled at 2 samples/bit, the Blackman window, and 50% overlap.

2.3.6 Symbol-by-Symbol Detector

A block diagram of the symbol-by-symbol detector is shown in Figure 2.11. Note that the detection filter is applied with the equalizer filter in Figure 2.6. Symbol-by-symbol detection comprises a detection filter operating at 2 samples/bit, a phase lock loop (PLL) operating at 1 sample/bit, and a decision device also operating at 1 sample/bit. Before the symbols are detected, the equalized samples $\hat{s}(n)$ are passed through the detection filter then down-sampled by 2. The

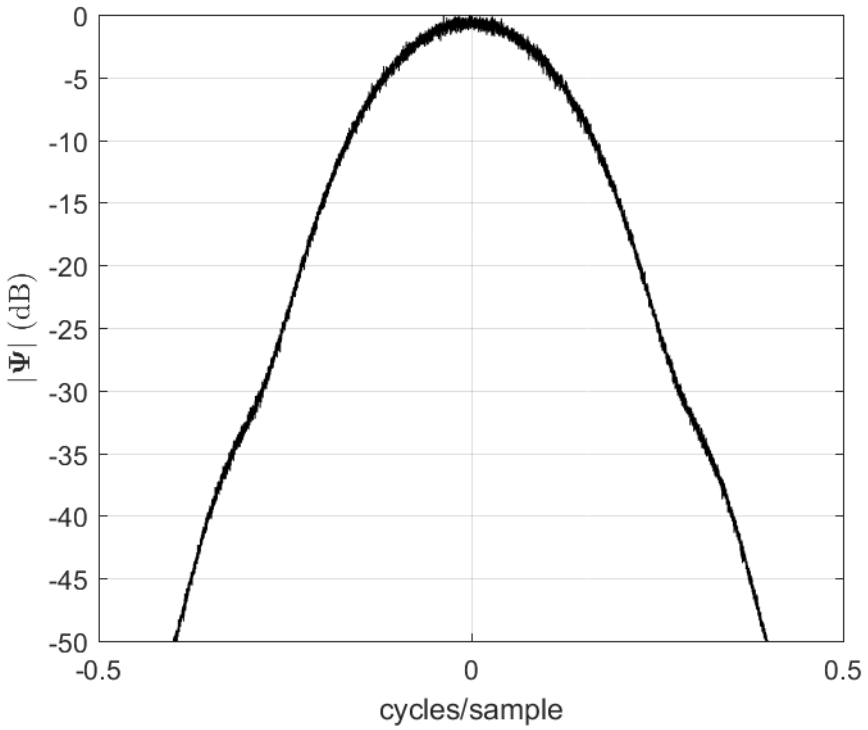


Figure 2.9: SOQPSK-TG power spectral density.

detection filter **d** is the length $L_{\mathrm{d}} = 23$ FIR filter whose response is shown in Figure 2.10 [14, Fig. 3]. The symbol-by-symbol detector block in Figure 2.6 is an OQPSK detector.

A phase lock loop (PLL) is needed in the OQPSK detector to track out residual frequency offset. The residual frequency offset results from a frequency offset estimation error. Equalizers mitigate the effects of phase offset, timing offset, and ISI because all of these impairments form the composite channel seen by the equalizer. A frequency offset is different, and cannot be mitigated by the equalizer alone. The PLL tracks out the residual frequency offset using a feedback control loop. The feedback control loop operates in a data-aided mode for $k < L_{\mathrm{p}} + L_{\mathrm{ASM}}$ using the known bits of preamble and ASM, denoted $a(k)$ in the figure. Note that $y(n)$ is indexed at 2 samples/bit while $y(k)$ and $\tilde{y}(k)$ are indexed at 1 sample/bit.

Implementing a PLL may not seem feasible in GPUs because the feedback loop cannot be parallelized. But the PAQ system processes 3104 packets of data simultaneously in parallel. Running the PLL and detector serially through a full packet of samples is relatively fast because the loop requires only 10 floating point operations and a few logic decisions.

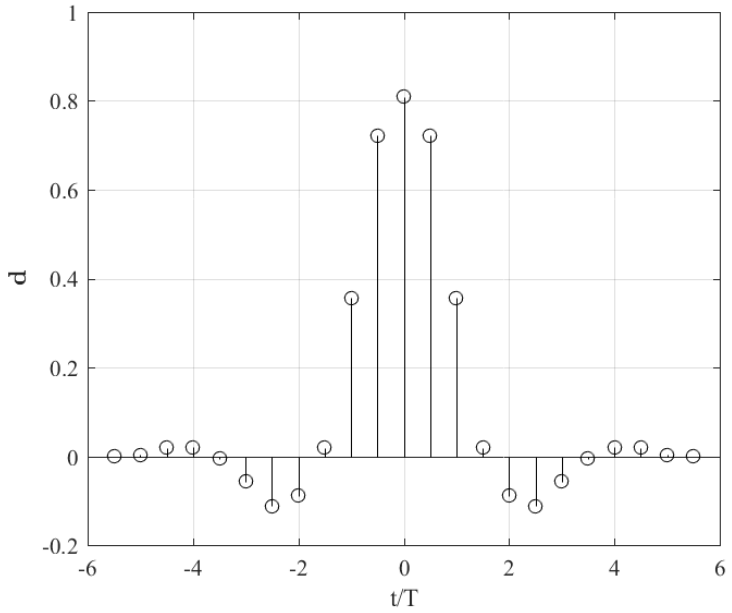


Figure 2.10: SOQPSK detection filter **d**.

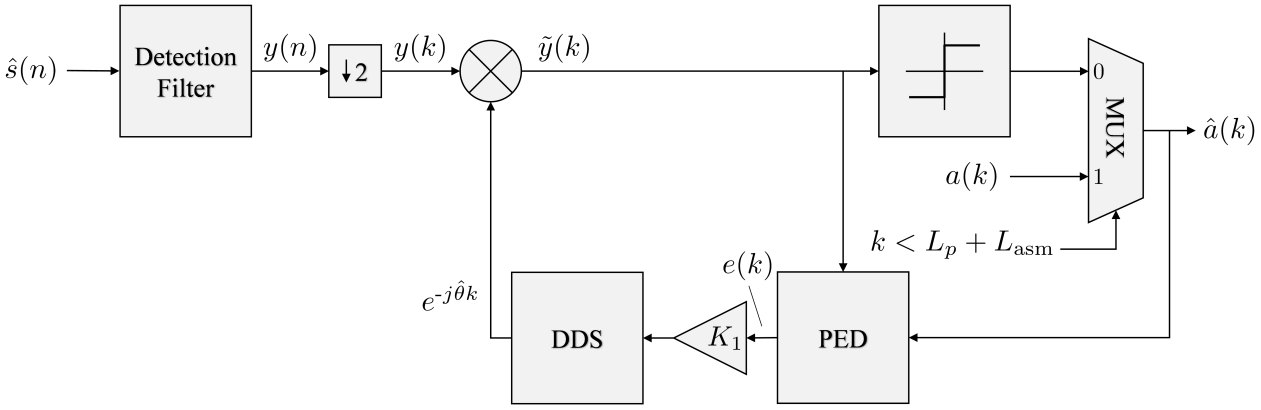


Figure 2.11: Offset Quadrature Phase Shift Keying symbol-by-symbol detector.

CHAPTER 3. SIGNAL PROCESSING IN GPUS

A graphics processing unit (GPU) is a computational unit with a highly-parallel architecture well-suited for executing the same function on many data elements. In the past, GPUs were used to process graphics data but in 2008 NVIDIA released the Tesla GPU. Telsa GPUs are built for general purpose high performance computing. Figure 3.1 shows the form factor of a Tesla K40c and K20c.

In 2007 NVIDIA released an extension to C, C++ and Fortran called CUDA (Compute Unified Device Architecture). CUDA enables GPUs to be used for high performance computing in computer vision, deep learning, artificial intelligence, and signal processing [15]. CUDA allows a programmer to write C++ like functions that are massively parallel called kernels. To invoke parallelism, a GPU kernel executed N times with the work distributed to N_{min} total threads that run concurrently. To achieve the full potential of high performance GPUs, kernels must be written with some basic concepts about GPU architecture and memory in mind. This chapter will show the following:

- Optimizing memory access leads to faster execution time, rather than optimizing number of floating point operations.
- The number of threads per block can significantly affect execution time.
- CPU and GPU processing can be pipelined.
- Convolution maps very well to GPUs using the Fast Fourier Transform (FFT).
- Batched processing leads to faster execution time per batch.

Figure 3.1: NVIDIA Tesla K40c and K20c.

3.1 GPU and CUDA Introduction

3.1.1 An Example Comparing CPU and GPU

If a programmer has some C++ experience, learning how to program GPUs using CUDA comes fairly easily. GPU code still runs top to bottom and memory still has to be allocated. The only real difference is the physical location of the memory and how functions run on GPUs. To run functions or kernels on GPUs, the memory must be copied from the host (CPU) to the device (GPU). Once the memory has been copied, parallel GPU kernels operate on the data. After GPU kernel execution, results are usually copied back from the device (GPU) to the host (CPU).

Listing 3.1 shows a simple program that implements real-valued float vector addition in a CPU and a GPU. The vector $\mathbf{C}_1$ is the sum of the vectors $\mathbf{A}_1$ and $\mathbf{B}_1$ computed in the CPU. The vector $\mathbf{C}_2$ is the sum of the vectors $\mathbf{A}_2$ and $\mathbf{B}_2$ computed in the GPU. Line 42 the CPU computes $\mathbf{C}_1$ by summing elements of $\mathbf{A}_1$ and $\mathbf{B}_1$ together sequentially. Figure 3.2 shows how the CPU se-

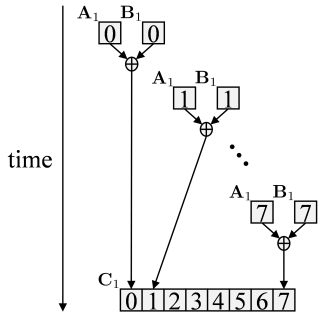


Figure 3.2: A block diagram of how a CPU sequentially performs vector addition.

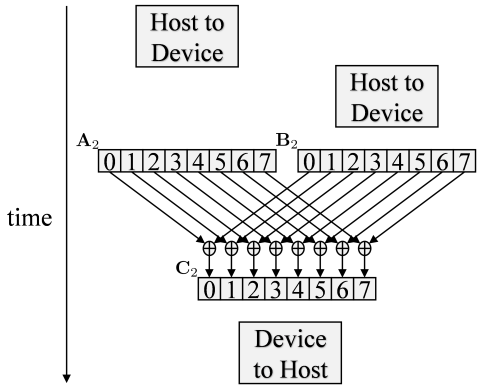


Figure 3.3: A block diagram of how a GPU performs vector addition in parallel.

quentially computes one element of $\mathbf{C}_1$ at time by summing one element from $\mathbf{A}_1$ and one element $\mathbf{B}_1$.

The GPU performs all the summations in parallel because each element of $\mathbf{C}_2$ is independent of all other elements. Before the computation of $\mathbf{C}_2$ can execute on the GPU, the vectors in host memory $\mathbf{A}_1$ and $\mathbf{B}_1$ are copied to device memory vectors $\mathbf{A}_2$ and $\mathbf{B}_2$ as shown on lines 60 and 61. Once $\mathbf{A}_2$ and $\mathbf{B}_2$ are on the GPU, the vector $\mathbf{C}_2$ is computed by calling the GPU kernel VecAddGPU on line 75. VecAddGPU computes all the elements of $\mathbf{C}_2$ by performing a summation of all the elements of $\mathbf{A}_2$ and $\mathbf{B}_2$. The vector $\mathbf{C}_2$ is then copied from device memory to host memory on line 78. Figure 3.3 shows how the GPU computes $\mathbf{C}_2$ in parallel.

Listing 3.1: Comparison of CPU and GPU code.

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
using namespace std;

void VecAddCPU(float* destination,float* source0,float* source1,int myLength){
        for(int i = 0; i < myLength; i++)
                destination[i] = source0[i] + source1[i];
}

__global__ void VecAddGPU(float* destination, float* source0, float* source1, int lastThread
    ){
        int i = blockIdx.x*blockDim.x + threadIdx.x;

        // don't access elements out of bounds
        if(i >= lastThread)
                return;

        destination[i] = source0[i] + source1[i];
}

int main(){
        int N = pow(2,22);
        cout << N << endl;
        /**
        * Vector Addition on CPU
        */
        // allocate memory on host
        float *A1;
        float *B1;
        float *C1;
        A1 = (float*) malloc (N*sizeof(float));
        B1 = (float*) malloc (N*sizeof(float));
        C1 = (float*) malloc (N*sizeof(float));

        // Initialize vectors 0-99
        for(int i = 0; i < N; i++){
                A1[i] = rand()%100;
                B1[i] = rand()%100;
        }

        // vector sum C1 = A1 + B1
        VecAddCPU(C1, A1, B1, N);

        /**
        * Vector Addition on GPU
        */
        // allocate memory on host for result
        float *C2;
        C2 = (float*) malloc (N*sizeof(float));

        // allocate memory on device for computation
        float *A2_gpu;
        float *B2_gpu;
        float *C2_gpu;
        cudaMalloc(&A2_gpu, sizeof(float)*N);
        cudaMalloc(&B2_gpu, sizeof(float)*N);
        cudaMalloc(&C2_gpu, sizeof(float)*N);

        // Copy vectors A and B from host to device
        cudaMemcpy(A2_gpu, A1, sizeof(float)*N, cudaMemcpyHostToDevice);
        cudaMemcpy(B2_gpu, B1, sizeof(float)*N, cudaMemcpyHostToDevice);

        // Set optimal number of threads per block
        int T_B = 32;

```

```
66          // Compute number of blocks for set number of threads
67          int B = N/T_B;
68
69          // If there are left over points, run an extra block
70          if(N % T_B > 0)
71                  B++;
72
73          // Run computation on device
74          //for(int i = 0; i < 100; i++)
75          VecAddGPU<<<B, T_B>>>(C2_gpu, A2_gpu, B2_gpu, N);
76
77          // Copy vector C2 from device to host
78          cudaMemcpy(C2, C2_gpu, sizeof(float)*N, cudaMemcpyDeviceToHost);
79
80          // Compare C2 to C1
81          bool equal = true;
82          for(int i = 0; i < N; i++)
83                  if(C1[i] != C2[i])
84                          equal = false;
85          if(equal)
86                  cout << "C2 is equal to C1." << endl;
87          else
88                  cout << "C2 is NOT equal to C1." << endl;
89
90          // Free vectors on CPU
91          free(A1);
92          free(B1);
93          free(C1);
94          free(C2);
95
96          // Free vectors on GPU
97          cudaFree(A2_gpu);
98          cudaFree(B2_gpu);
99          cudaFree(C2_gpu);
100 }
```

3.1.2 GPU Kernel Using Threads and Thread Blocks

A GPU kernel is executed by launching blocks with a set number of threads per block. In the Listing 3.1, VecAddGPU is launched on line 75 with 32 threads per block. The total number of threads launched on the GPU is the number of blocks times the number of threads per block. VecAddGPU needs to be launched with at least $N = 2^{22}$ (line 22) threads or $2^{22}/32$ blocks of 32 threads.

CUDA gives each thread launched in a GPU kernel a set of unique indices called threadIdx and blockIdx. threadIdx is the thread index inside the assigned thread block. blockIdx is the index of the block to which the thread is assigned. Both threadIdx and blockIdx are three dimensional (i.e., they both have x, y, and z components). In this thesis only the x dimension is used, because the GPU kernels operate only on one dimensional vectors. blockDim is the number of threads assigned

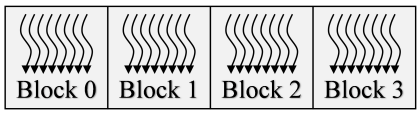


Figure 3.4: 32 threads launched in 4 thread blocks with 8 threads per block.

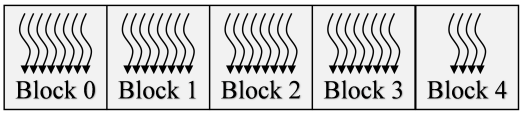


Figure 3.5: 36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.

per block, in fact blockDim is equal to the number of threads per block because the vectors are one dimensional.

To convert the CPU "for loop" on line 7 to a GPU kernel, at least N threads are launched with T threads per thread block. The number of blocks needed is $B = \frac{N}{T_B}$ or $B = \frac{N}{T} + 1$, if N is not an integer multiple of T. Figure 3.4 shows $N = 32$ threads launched in $B = 4$ thread blocks with $T = 8$ threads per block. Figure 3.5 shows $N = 36$ threads launched in $B = 5$ thread blocks with $T = 8$ threads per block. An full extra thread block is launched with $T = 8$ threads, but 4 threads are idle. Thread blocks are executed independent of other thread blocks. The GPU does not guarantee Block 0 will execute before Block 2.

3.1.3 GPU Memory

GPUs have plenty of computational resources, but most GPU kernels are limited by memory bandwidth to feed the computational units. GPU kernels execute faster if the kernel is designed to access memory efficiently, rather than reducing the computational burden. NVIDIA GPUs have many different types of memory to maximize speed and efficiency.

The fastest memory is private local memory, in the form of Registers and L1 Cache/shared memory. Local memory is fast, but only kilobytes are available. The slowest memory is public memory in the form of the L2 Cache and Global Memory. Public memory is slow, but gigabytes

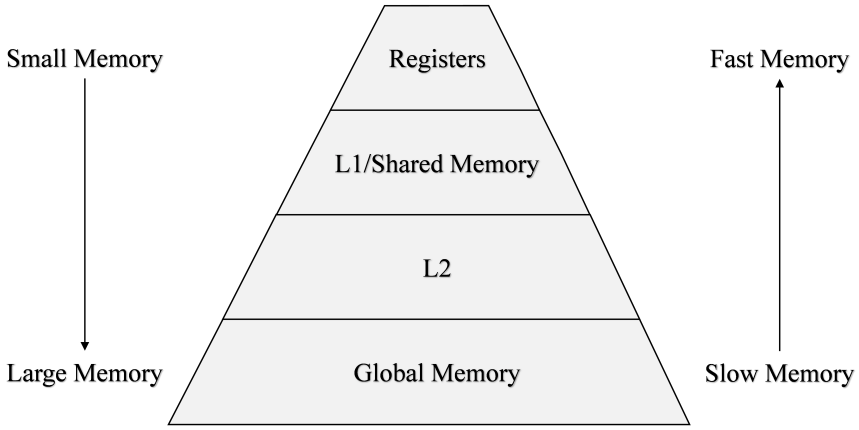


Figure 3.6: Diagram comparing memory size and speed. Global memory is massive but extremely slow. Registers are extremely fast but there are very few.

are available. Figure 3.6 shows the trade-off of memory speed and the size of different types of memory.

Figure 3.7 shows a picture of the GPU hardware. The solid boxes show that the L2 cache and Global Memory are physically located off the GPU chip. The dashed box shows that Registers and L1 Cache/Shared Memory are physically located *on* the GPU chip. A public memory access takes over 60 clock cycles, because the memory is off chip. A local memory access is only a few clock cycles because the memory is on chip.

Figure 3.8 illustrates where each type of memory is located. Threads have access to their own Registers and the L1 Cache. Threads in a block can coordinate using shared memory, because shared memory is private to the thread block. All threads have access to the L2 Cache and Global Memory. The figure also shows that thread blocks are assigned to streaming multiprocessors (SMs). CUDA handles all the thread block assignments to SMs. Table 3.1 lists Telsa K40c and K20c resources.

3.1.4 Thread Optimization

Most resources listed in Table 3.1 show how much memory per thread block is available. The number of threads per block and the amount of resources available have an inverse relationship. Threads have very little memory resources available, if a GPU kernel launches 1024 threads per

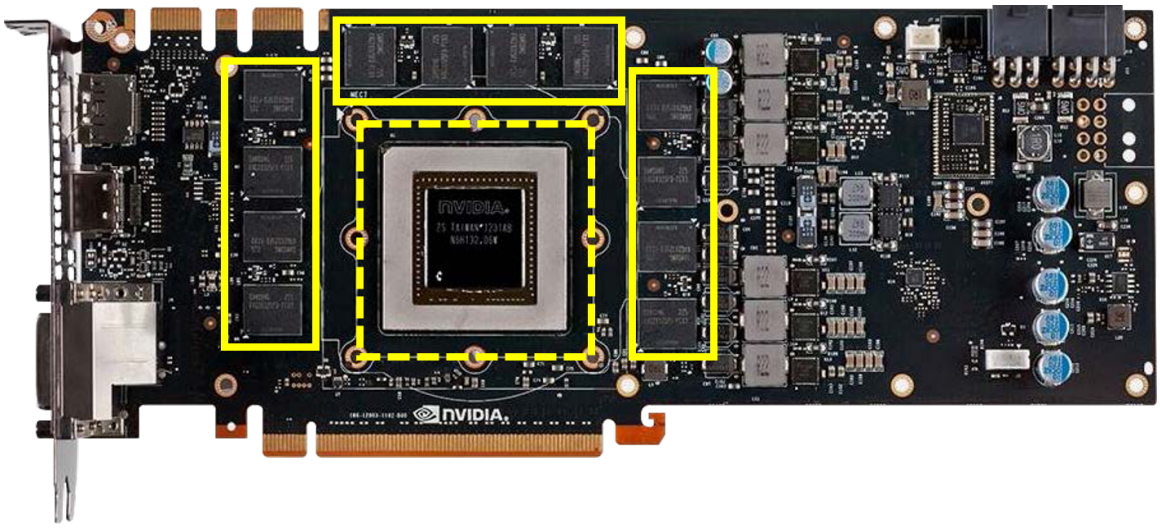


Figure 3.7: Example of an NVIDIA GPU card. The GPU chip with registers and L1/shared memory is shown in the dashed box. The L2 cache and global memory is shown off chip in the solid boxes.

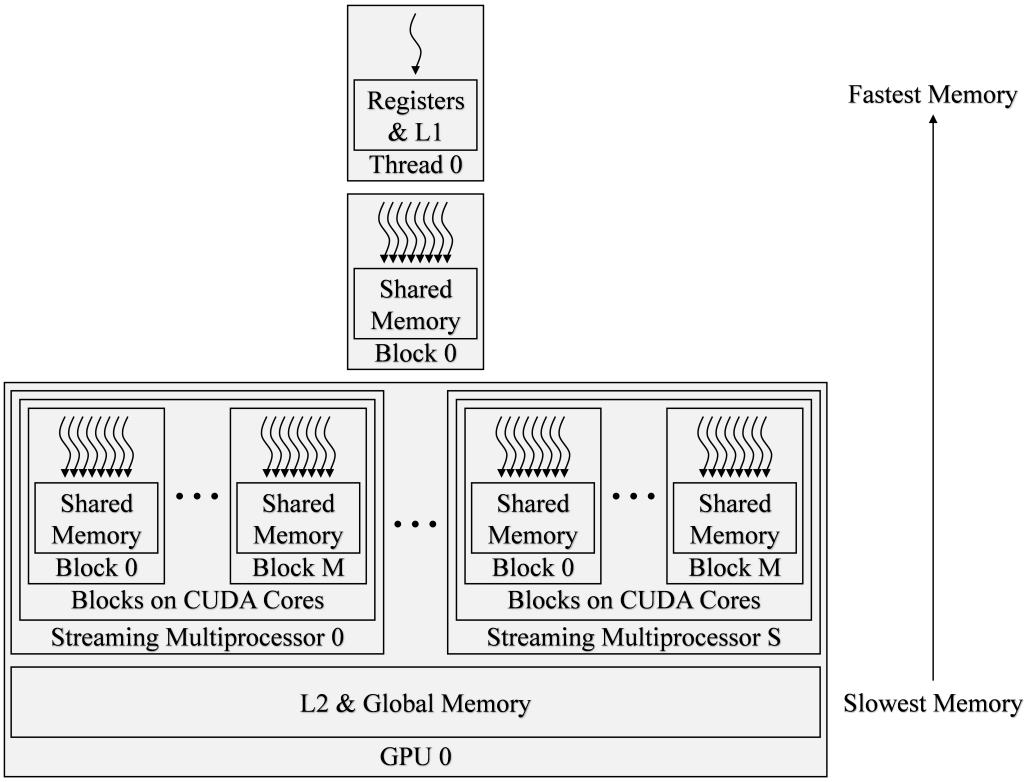


Figure 3.8: A block diagram where local, shared, and global memory is located. Each thread has private local memory. Each thread block has private shared memory. The GPU has global memory that all threads can access.

Table 3.1: The resources available with three NVIDIA GPUs used in this thesis (1x Tesla K40c 2x Tesla K20c). Note that CUDA configures the size of the L1 cache needed.

Feature	Per	Tesla K40c	Tesla K20c
Global Memory	GPU	12 GB	5 GB
L2 Cache Size	GPU	1.6 GB	1.3 GB
Memory Bandwidth		288 GB/s	208 GB/s
Shared Memory	Thread Block	49 kB	49 kB
L1 Cache Size	Thread Block	variable	variable
Registers	Thread Block	65536	65536
Maximum Threads	Thread Block	1024	1024
CUDA Cores	GPU	2880	2496
Base Core Clock		745 MHz	732 MHz

block. Threads have a lot of memory resources available, if a GPU kernel launches 32 threads per block. This section shows that finding the optimum number of threads per block can dramatically speed up GPU kernels.

Improving memory accesses should always be the first optimization, when a GPU kernel needs to be faster. The next step is to find the optimal number of threads per block to launch. Knowing the perfect number of threads per block to launch is challenging to calculate. Fortunately, the maximum number of possible threads per block is 1024 in the Tesla K40c and K20c GPUs. Listing 3.2 shows a simple test program that measures GPU kernel execution time, while varying the number of possible threads per block. The number of threads per block with the fastest computation time is the optimal number of threads per block for that specific GPU kernel.

Listing 3.2: Code snippet for thread optimization.

```
1  float milliseconds_opt = pow(2,10); // initiaize to "big" number
2  int T_B_opt;
3  int minNumTotalThreads = pow(2,20); // set to minimum number of required threads
4  for(int T_B = 1; T_B<=1024; T_B++){
5          int B = minNumTotalThreads/T_B;
6          if(minNumTotalThreads % T_B > 0)
7                  B++;
8          cudaEvent_t start, stop;
9          cudaEventCreate(&start);
10         cudaEventCreate(&stop);
11         cudaEventRecord(start);
12
13         GPUkernel<<<B, T_B>>>(dev_vec0, dev_vec1);
```

```
14
15          cudaEventRecord(stop);
16          cudaEventSynchronize(stop);
17          float milliseconds = 0;
18          cudaEventElapsedTime(&milliseconds, start, stop);
19          cudaEventDestroy(start);
20          cudaEventDestroy(stop);
21          if(milliseconds<milliseconds_opt){
22                  milliseconds_opt = milliseconds;
23                  T_B_opt = T_B;
24          }
25  }
26  cout << "Optimal Threads Per Block " << T_B_opt << endl
27  cout << "Optimal Execution Time    " << milliseconds_opt << endl;
```

Most of the time the optimal number of threads per block is a multiple of 32 this is because at the lowest level of architecture, GPUs perform computations in warps. Warps are groups of 32 threads that perform every computation together in lock step. If the number of threads per block is not a multiple of 32, some threads in a warp are idle and the GPU has unused computational resources.

Figure 3.9 shows the execution time of an example GPU kernel. The optimal execution time is 0.1078 ms at the optimal 96 threads per block. By simply adjusting the number of threads per block, the execution time of this example kernel can be reduced by a factor of 2.

Adjusting the number of threads per block does not always dramatically reduce execution time. Figure 3.10 shows the execution time for another GPU kernel with varying threads per block. The execution time of this example kernel can be reduced by 1.12 by launching 560 threads per block.

While designing a custom GPU kernel to obtain a major speed up is satisfying, CUDA has optimized GPU libraries that are extremely useful and efficient with exceptional documentation. The CUDA libraries are written by NVIDIA engineers to maximize the performance of NVIDIA GPUs. The libraries explained in this thesis include cuFFT, cuBLAS and cuSolverSp.

3.1.5 CPU and GPU Pipelining

While GPU kernels execute physically on the GPU, the GPU only executes instructions received from the host CPU. The CPU is idle while it waits for GPU kernels to execute. To

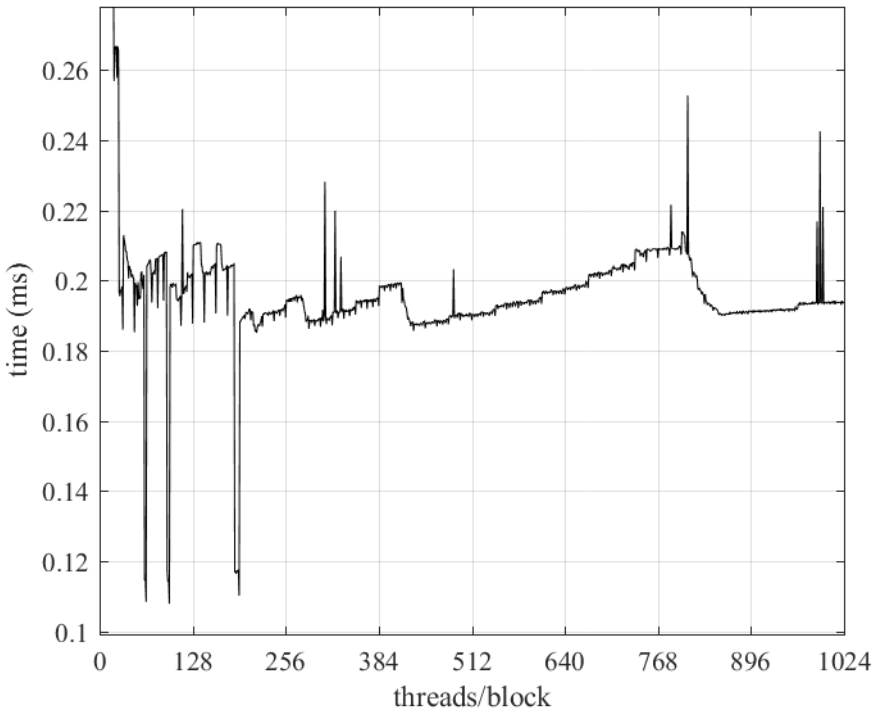


Figure 3.9: Plot showing how execution time is affected by changing the number of threads per block. The optimal execution time for an example GPU kernel is 0.1078 ms at the optimal 96 threads per block.

introduce CPU and GPU pipelining, the CPU can be pipelined by performing other operations while waiting for the GPU to finish executing kernels.

A basic CPU GPU program with no pipelining is shown in Listing 3.3. The CPU acquires data from myADC on Line 5. After the CPU takes time to acquire data, the data is copied from the host (CPU) to the device (GPU) on line 8. The data is processed on the GPU once then the result is copied back to the device to host on line 9 and 10. The cudaDeviceSynchronize function on line 13 blocks CPU until all GPU instructions are finished executing. Note that the CPU is blocked during any host to device or device to host transfer. Acquiring and copying data takes processing time on the CPU and GPU. Figure 3.11 shows a block diagram of what is happening on the CPU and GPU in Listing 3.3 (end of the section). The GPU is idle while the CPU is acquiring data and the CPU is idle while the GPU is processing.

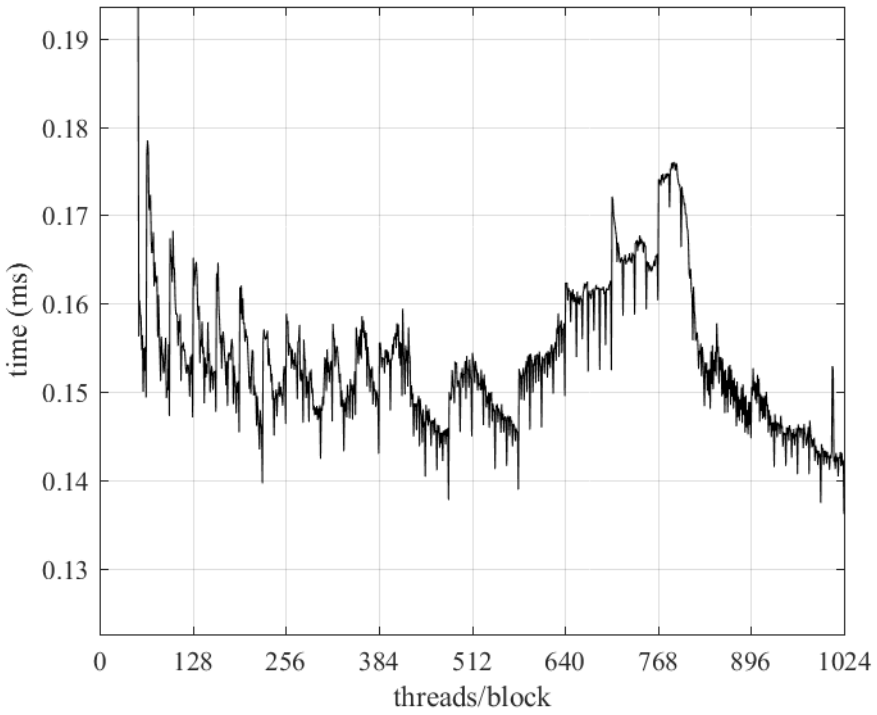


Figure 3.10: Plot showing the number of threads per block doesn't always drastically affect execution time.

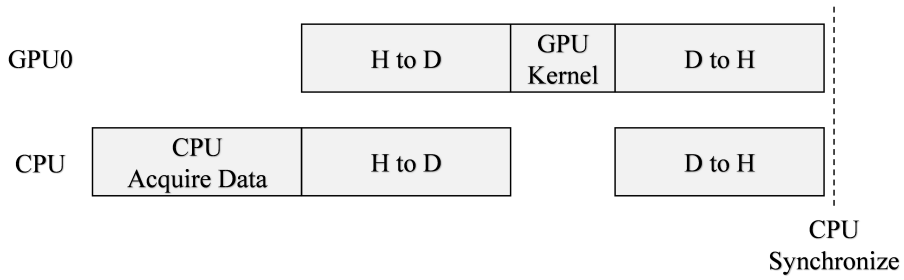


Figure 3.11: The typical approach of CPU and GPU operations. This block diagram shows the profile of Listing 3.3.

Listing 3.4 (end of the section) shows how to CPU and GPU operations can be pipelined. Assuming data is already on the GPU from a prior computation, the CPU gives processing instructions to the GPU then acquires data. The CPU then does an asynchronous data transfer to a temporary vector on the GPU. The GPU first performs a device to device transfer from the tem-

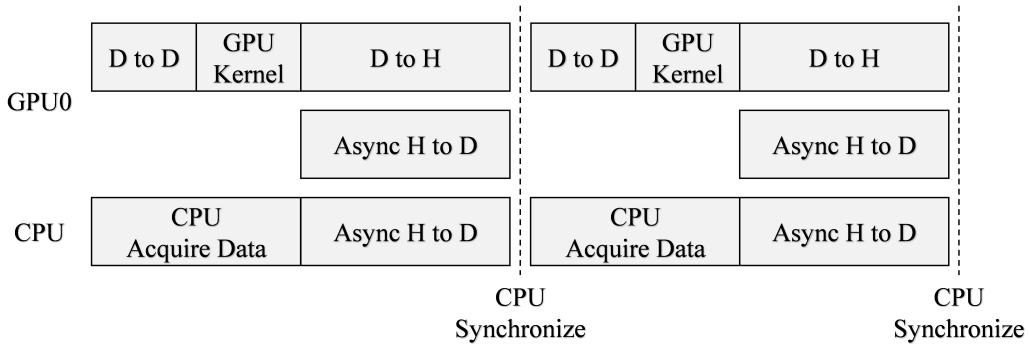


Figure 3.12: GPU and CPU operations can be pipelined. This block diagram shows a profile of Listing 3.4.

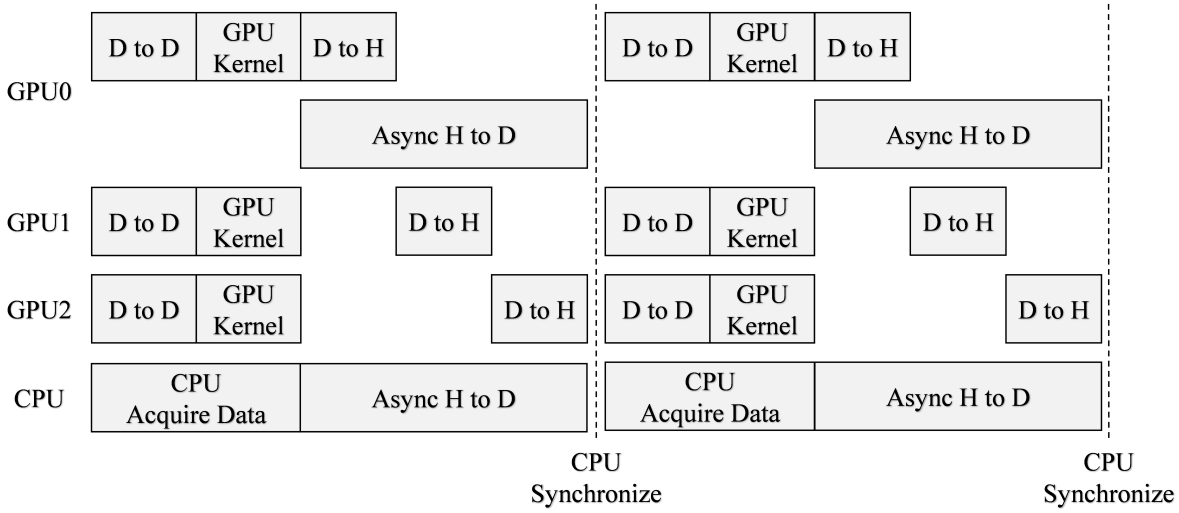


Figure 3.13: A block diagram of pipelining a CPU with three GPUs.

porary vector. The GPU then runs the GPUkernel and transfers the result to the host. Note that device to device transfers do not block the CPU. This system suffers a full cycle latency.

Pipelineing can be extended to multiple GPUs for even more throughput, but only suffer latency of copying memory to one GPU. Figure 3.13 shows a block diagram of how three GPUs can be pipelined. A strong understanding of the full system is required to pipeline at this level.

Listing 3.3: Example code Simple example of the CPU acquiring data from myADC, copying from host to device, processing data on the device then copying from device to host. No processing occurs on device while CPU is acquiring data.

```
int main()
{
        ...
        // CPU Acuire Data
        myADC.acquire(vec);

        // Launch instructions on GPU
        cudaMemcpy(dev_vec0, vec,      numBytes, cudaMemcpyHostToDevice);
        GPUkernel<<<1, N>>>(dev_vec0);
        cudaMemcpy(vec,      dev_vec0, numBytes, cudaMemcpyDeviceToHost);

        // Synchronize CPU with GPU
        cudaDeviceSynchronize();
        ...
}
```

Listing 3.4: Example code Simple of the CPU acquiring data from myADC, copying from host to device, processing data on the device then copying from device to host. No processing occurs on device while CPU is acquiring data.

```
int main()
{
        ...
        // Launch instructions on GPU
        cudaMemcpy(dev_vec, dev_temp, numBytes, cudaMemcpyDeviceToDevice);
        GPUkernel<<<N, M>>>(dev_vec);
        cudaMemcpy(vec,     dev_vec,  numBytes, cudaMemcpyDeviceToHost);

        // CPU Acuire Data
        myADC.acquire(vec);
        cudaMemcpyAsync(dev_temp, vec, numBytes, cudaMemcpyHostToDevice);

        // Synchronize CPU with GPU
        cudaDeviceSynchronize();
        ...

        ...
        // Launch instructions on GPU
        cudaMemcpy(dev_vec, dev_temp, numBytes, cudaMemcpyDeviceToDevice);
        GPUkernel<<<N, M>>>(dev_vec);
        cudaMemcpy(vec,     dev_vec,  numBytes, cudaMemcpyDeviceToHost);

        // CPU Acuire Data
        myADC.acquire(vec);
        cudaMemcpyAsync(dev_temp, vec, numBytes, cudaMemcpyHostToDevice);

        // Synchronize CPU with GPU
        cudaDeviceSynchronize();
        ...
}
```

3.2 GPU Convolution

Convolution is one of the most important tools in digital signal processing. The PAQ system explained uses convolution up to 26 times per packet, depending on the number of CMA iterations. If convolution execution time can be reduced by 10 ms, the full system execution time can reduced by 260 ms. This section will use the following notation:

- The signal **x** is a vector of N complex samples indexed by $x(n)$ where, $0 \leq\leq N-1$.
- The filter **h** is a vector of L complex samples indexed by $h(n)$ where, $0 \leq\leq L-1$.
- The filtered signal **y** is a vector resulting from the convolution of **x** and **h**. **y** is $C = N+L-1$ complex samples and is indexed by $y(n)$ where, $0 \leq n \leq C-1$.
- The forward Fast Fourier Transform (FFT) of the vector **x** is denoted $\mathscr{F}(\mathbf{x})$.
- The inverse Fast Fourier Transform (IFFT) of the vector **x** is denoted $\mathscr{F}^{-1}(\mathbf{x})$.

Discrete time convolution applies the filter **h** to the signal **x** resulting in the filter signal **y**. Convolution in the time domain is

$$y(n) = \sum_{m=0}^{L-1} x(m)h(n-m), \tag{3.1}$$

and the frequency domain is

$$\mathbf{y} = \mathscr{F}^{-1}(\mathscr{F}(\mathbf{x}) \times \mathscr{F}(\mathbf{h})). \tag{3.2}$$

Figure 3.14 shows block diagrams for time-domain and frequency domain convolution. This section will show:

- GPU convolution is faster than CPU convolution with large data sets using execution time as a metric.
- GPU convolution execution time is dependent more on memory access than floating point operations.

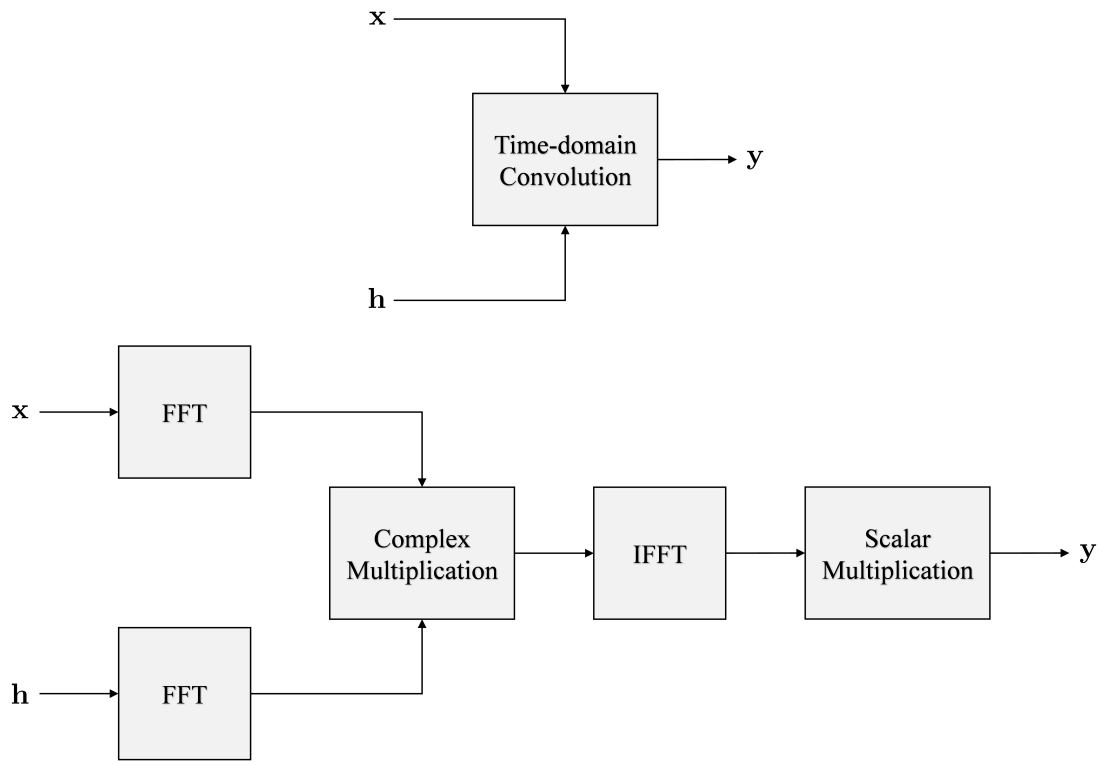


Figure 3.14: Block diagrams showing time-domain convolution and frequency-domain convolution.

- Performing batched GPU convolution invokes more parallelism and decreases execution time per batch.
- Batched GPU frequency-domain convolution executes faster than batched GPU time-domain convolution.

3.2.1 Floating Point Operation Comparison

Traditionally the number of floating point operations (flops) is used to estimate how computationally intense an algorithm is. Each complex multiplication

$$(A + jB) \times (C + jD) = (AC - BD) + j(AD + BC), \tag{3.3}$$

requires 6 flops, 4 multiplications and 2 additions/subtractions. Output elements of $\mathbf{y}$ in Equation (3.1) requires $8L = (6+2)L$ flops, 2 extra flops are required for each summand. The time-domain convolution requires

$$8LC \text{ flops}, \tag{3.4}$$

where $C = N + L - 1$ is the length of the convolution result.

To leverage the Cooley-Tukey radix 2 Fast Fourier Transform (FFT) in frequency-domain convolution, common practice is to compute the M point FFT where $M = 2^u$ and $u = \lceil \log_2(C) \rceil$. Both the CPU based Fastest Fourier Transform in the West (FFTW) library and the NVIDIA GPU cuFFT library use the Cooley-Tukey radix 2 FFT. Each FFT or IFFT requires $5M\log_2(M)$ flops [16, 17]. As shown by Equation (3.2), frequency-domain convolution requires

$$3 \times 5M\log_2(M) + 6M \text{ flops}, \tag{3.5}$$

from 3 FFTs and M point-to-point multiplications.

Sections 2.2 and 2.3.6 show the PAQ system has one signal length, $N = L_{\text{pkt}} = 12{,}672$ samples and two filter lengths $L = L_{\text{df}} = 23$ and $L = L_{\text{eq}} = 186$. Figures 3.15 through 3.17 compare the number of flops required for time-domain and frequency-domain convolution. The figures compare flops by fixing the signal length with variable filter length or visa versa. These figures show applying a 186 tap filter to a 12,672 sample signal requires less flops in the frequency domain and applying a 23 tap filter to a 12,672 sample signal requires less flops in the time domain.

3.2.2 CPU and GPU Single Convolution Using Batch Processing Comparison

This section will show GPU convolution execution time is dependent more on memory access than the number of required floating point operations, while CPU convolution execution time is dependent on the number of floating point operations. To illustrate these points, the execution time of the code in Listing 3.5 (at the end of the chapter) was measured. The code implements convolution five different ways:

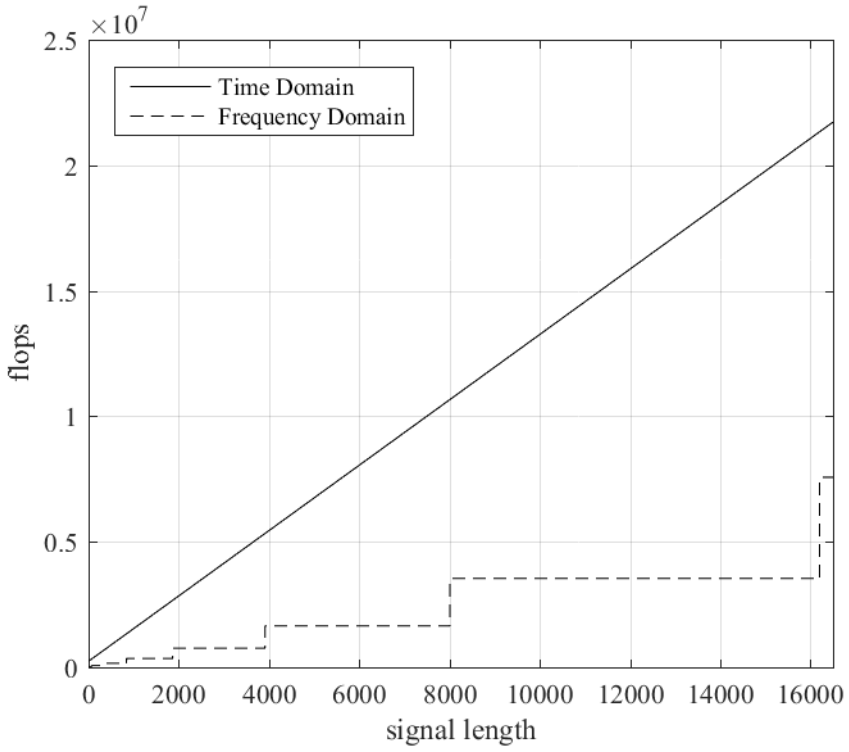


Figure 3.15: Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 186 tap complex filter.

- time-domain convolution in a CPU,
- frequency-domain convolution in a CPU using the FFTW library,
- time-domain convolution in a GPU using global memory,
- time-domain convolution in a GPU using shared memory, and
- frequency-domain convolution in a GPU using the cuFFT library.

The three time-domain convolution implementations compute (3.1) directly. The two frequency-domain convolution implementations compute (3.2), using the CPU FFTW library and the GPU based cuFFT library. The cuFFT library uses global memory and shared memory to be as fast and efficient as possible. For a given signal and filter length, a good CUDA programmer can make an educated guess on which algorithm is faster. There is no clear conclusion, until all the algorithms have been implemented and measured.

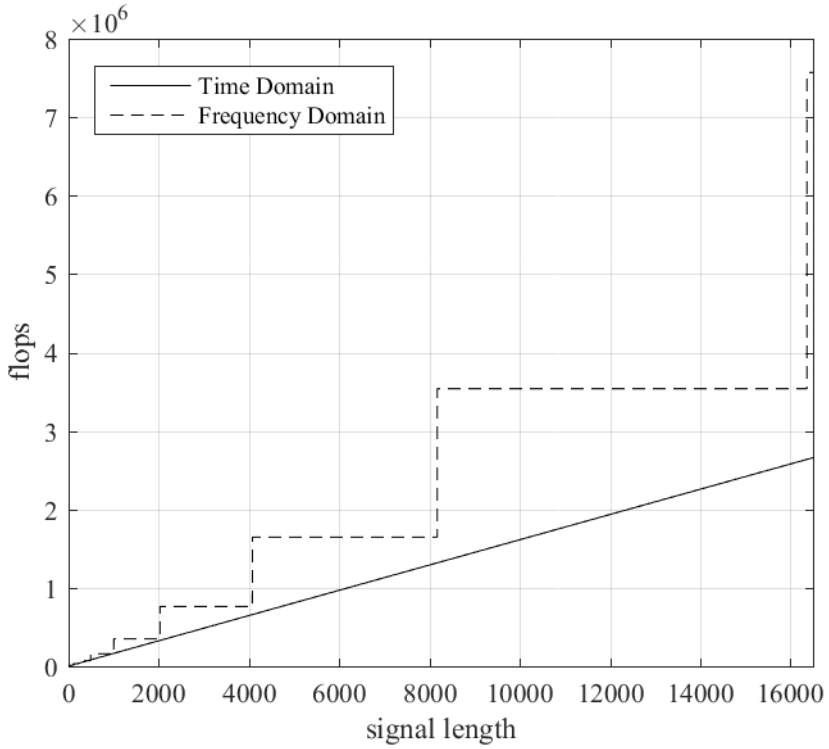


Figure 3.16: Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 23 tap complex filter.

Table 3.2: Defining start and stop lines for timing comparison in Listing 3.5.

Algorithm	Function	Start Line	Stop Line
CPU time domain	ConvCPU	208	210
CPU frequency domain	FFTW	213	259
GPU time domain global	ConvGPU	267	278
GPU time domain shared	ConvGPUshared	282	293
GPU frequency domain	cuFFT	301	327

All the memory transfers to and from the GPU were timed for a fair comparison of GPU to CPU execution time. Table 3.2 shows how the execution time was measured for each convolution implementation. Figures 3.18 through 3.21 compare execution time of the five different convolution implementations by fixing the filter length with variable signal length or vise versa. Sub-windows emphasize points that are of interest to the PAQ system. The variations in the time-

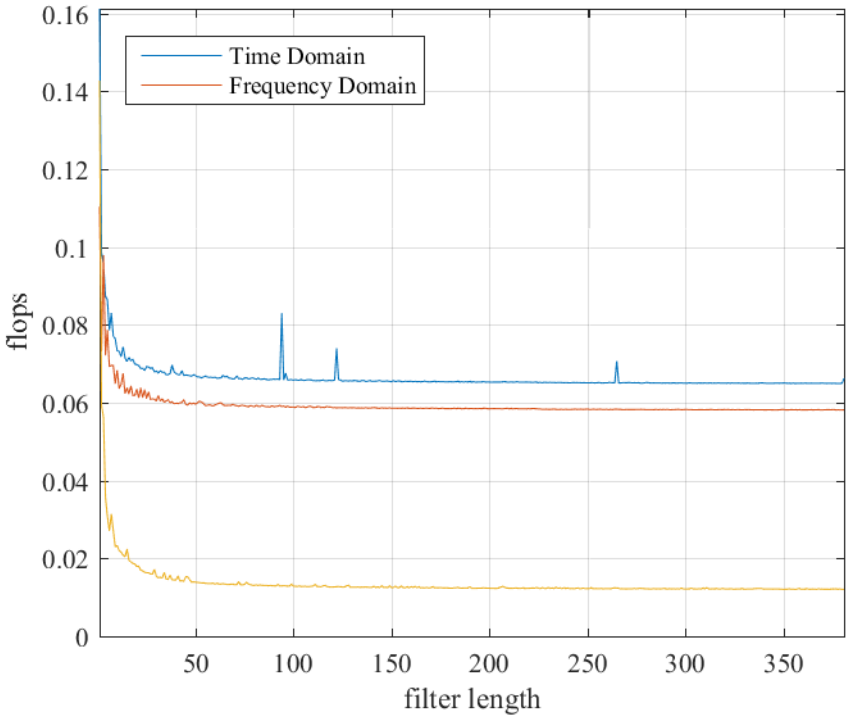


Figure 3.17: Comparison of number of floating point operations (flops) required to convolve a 12,672 sample complex signal with a variable length tap complex filter.

domain CPU execution times are due to the claims on the host CPU resources by the operating system. To clean up the time samples, local minima were found in windows ranging from 3 to 15 samples. The smallest windows possible were used to produce the results.

Comparing Figures 3.19 through 3.21 to Figures 3.15 through 3.17 shows CPU and GPU convolution have the same structure that the number of flops predicted, except GPU convolution is not affected as much by varied signal or filter lengths. The convolution execution time comparison demonstrates the observation that most GPU kernels execution time is limited by memory bandwidth not computational resources. Tables 3.3 and 3.4 show the GPU time-domain algorithm using shared memory is fastest for the signal length and filter lengths of the PAQ system, when performing a single complex convolution.

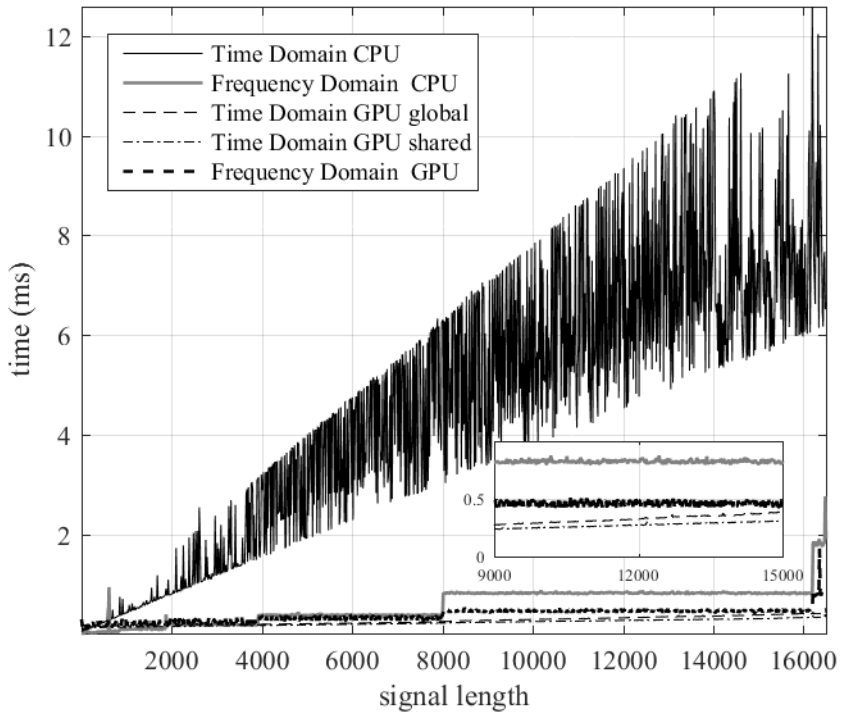


Figure 3.18: Comparison of a complex convolution on CPU and GPU. The signal length is variable and the filter is fixed at 186 taps. The comparison is messy without lower bounding.

Table 3.3: Convolution computation times with signal length 12,672 and filter length 186 on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
CPU time domain	ConvCPU	6.2683
CPU frequency domain	FFTW	0.8519
GPU time domain global	ConvGPU	0.3467
GPU time domain shared	ConvGPUshared	0.2857
GPU frequency domain	cuFFT	0.4490

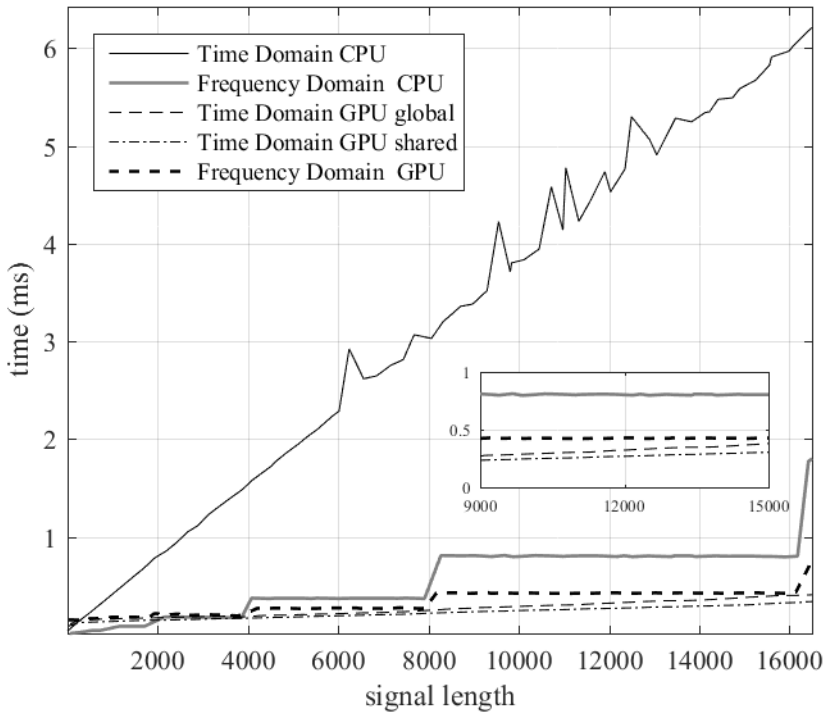


Figure 3.19: Comparison of a complex convolution on CPU and GPU. The signal length is variable and the filter is fixed at 186 taps. A lower bound was applied by searching for a local minima in 15 sample width windows.

Table 3.4: Convolution computation times with signal length 12,672 and filter length 23 on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
CPU time domain	ConvCPU	0.6429
CPU frequency domain	FFTW	0.8899
GPU time domain global	ConvGPU	0.2406
GPU time domain shared	ConvGPUshared	0.2346
GPU frequency domain	cuFFT	0.3231

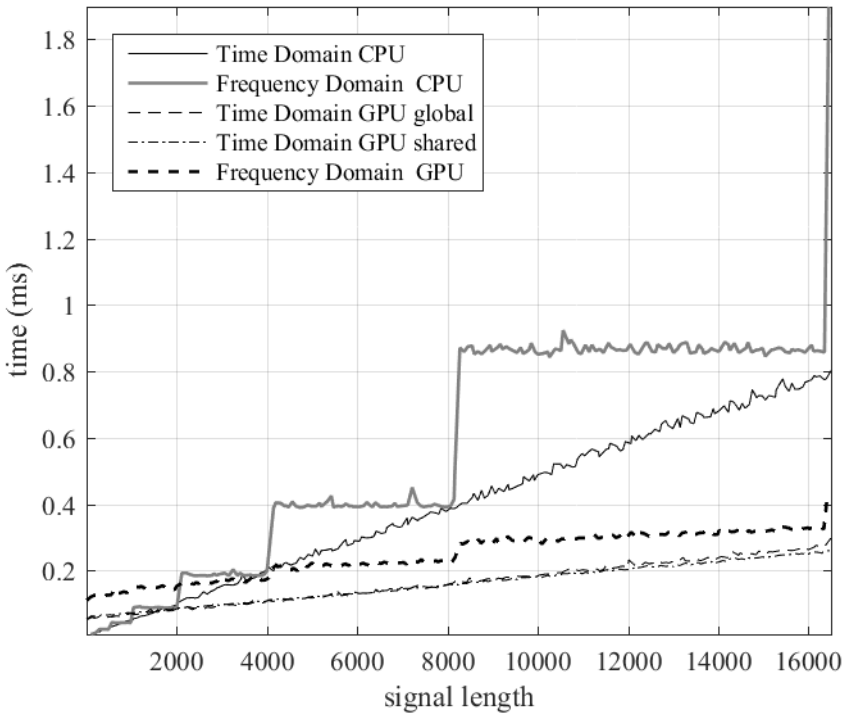


Figure 3.20: Comparison of a complex convolution on CPU and GPU. The signal length is variable and the filter is fixed at 23 taps. A lower bound was applied by searching for a local minima in 5 sample width windows.

3.2.3 Convolution Using Batch Processing

Section 3.2.2, illustrated convolving one signal with one filter, does not leverage the full power of parallel processing in GPUs. The received signal in the PAQ system has a packetized structure with 3104 packets per 1907 ms. Rather than processing each packet separately, the packets may be buffered and processed in a batch. Batch processing in GPUs has less CPU overhead and introduces an extra level of parallelism. Batch processing has faster execution time per packet, than processing packets separately. CUDA has many libraries that have batch processing, including cuFFT, cuBLAS and cuSolverSp. Haidar et al. [18] showed batched libraries achieve more Gflops, than calling GPU kernels multiple times. Listing 3.6 (at the end of the chapter) shows three GPU implementations of convolution using batch processing and Table 3.5 shows how the execution time of the code was measured.

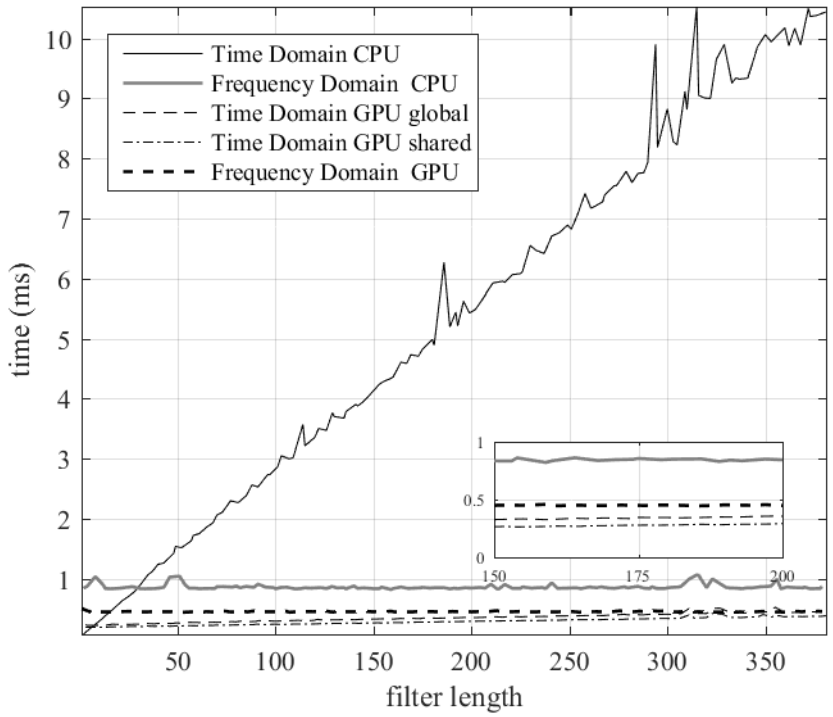


Figure 3.21: Comparison of a complex convolution on CPU and GPU. The filter length is variable and the signal is fixed at 12,672 samples. A lower bound was applied by searching for a local minima in three sample width windows.

Table 3.5: Defining start and stop lines for execution time comparison in Listing 3.6.

Algorithm	Function	Start Line	Stop Line
GPU time domain global	ConvGPU	197	204
GPU time domain shared	ConvGPUshared	212	219
GPU frequency domain	cuFFT	227	245

Figure 3.22 compares execution time of convolution using batch processing as the number of packets increases, note that no lower bounding was used. This figure shows that frequency-domain convolution leverages batch processing better than time-domain convolution. As expected, CPU-based convolution using batch processing is not competitive with GPU-based convolution using batch processing and thus CPU-batched processing is not explored any further.

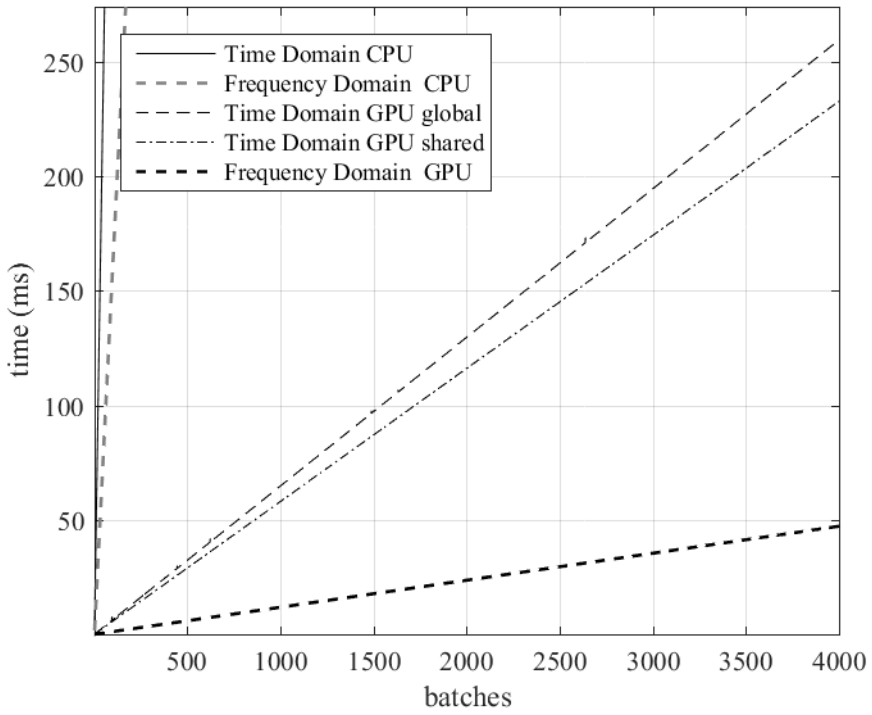


Figure 3.22: Comparison of a batched complex convolution on a CPU and GPU. The number of batches is variable while the signal and filter length is set to 12,672 and 186.

Now that the GPU and CPU execution time is not being compared, Table 3.5 shows execution times include only GPU kernels and exclude memory transfers. Figure 3.23 compares GPU convolution using batch processing execution time per batch as the number of packets increases. The figure shows execution time per batch decreases as the number of packets increases but stops improving after 70 packets.

Figures 3.24 through 3.26 compare execution time of the three GPU convolution implementations by fixing the filter length with variable signal length or vise versa. Tables 3.6 and 3.7 show the execution times for the signal length and filter lengths of the PAQ system when performing convolution using batch processing. Frequency-domain convolution using batch processing is fastest for 186 tap filters while time-domain convolution using batch processing and shared memory is fastest for 23 tap filters.

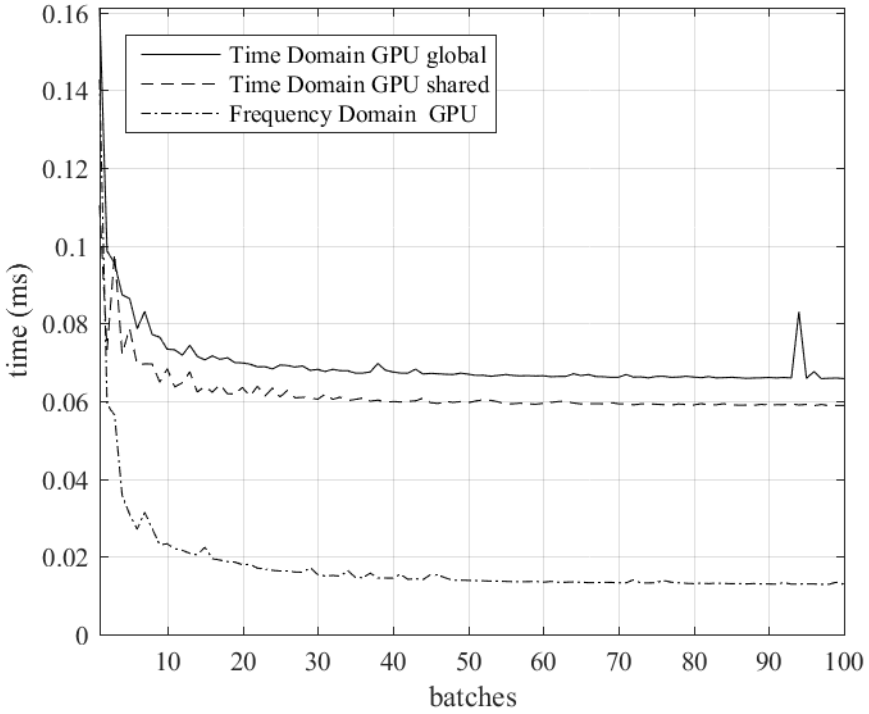


Figure 3.23: Comparison on execution time per batch for complex convolution. The number of batches is variable while the signal and filter length is set to 12,672 and 186.

Table 3.6: Convolution using batch processing execution times with for a 12,672 sample signal and 186 tap filter on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
GPU time domain global	ConvGPU	201.3
GPU time domain shared	ConvGPUshared	180.3
GPU frequency domain	cuFFT	36.8

Table 3.7: Convolution using batch processing execution times with for a 12,672 sample signal and 23 tap filter on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
GPU time domain global	ConvGPU	29.5
GPU time domain shared	ConvGPUshared	22.7
GPU frequency domain	cuFFT	39.0

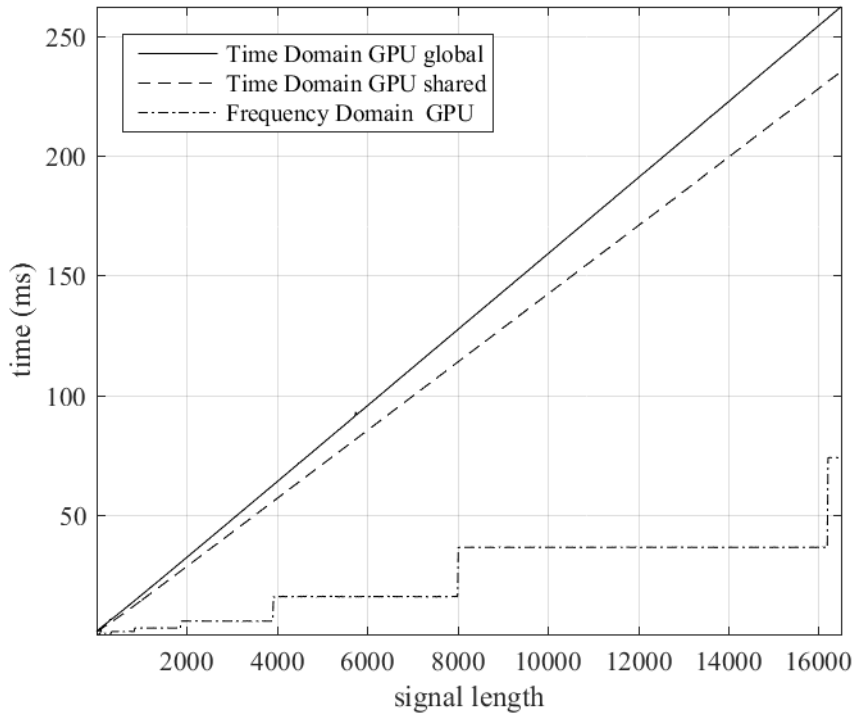


Figure 3.24: Comparison of complex convolution using batch processing on a GPU. The signal length is variable and the filter is fixed at 186 taps.

Until now, convolving one signal with only one filter has been considered. Figure 2.6 showed the received signal is filtered by two cascaded filters: an equalizer filter and a detection filter. The block diagrams in Figure 3.27 show the steps required for cascading time-domain and frequency-domain convolution.

Comparing the block diagrams in Figures 3.27 and 3.14, cascading two filters in the frequency domain only requires an extra FFT and point-by-point complex multiplication, while cascading filters in the time domain requires two time-domain convolutions. The first time-domain convolution produces a composite filter from the convolution of the 186 sample equalizer filter with the 23 tap detection filter. The second time-domain convolution applies the composite $208 = 186 + 23 - 1$ tap filter to a 12,672 sample signal. Table 3.8 shows the execution times for the signal length and filter lengths of the PAQ system, when performing cascaded convolution us-

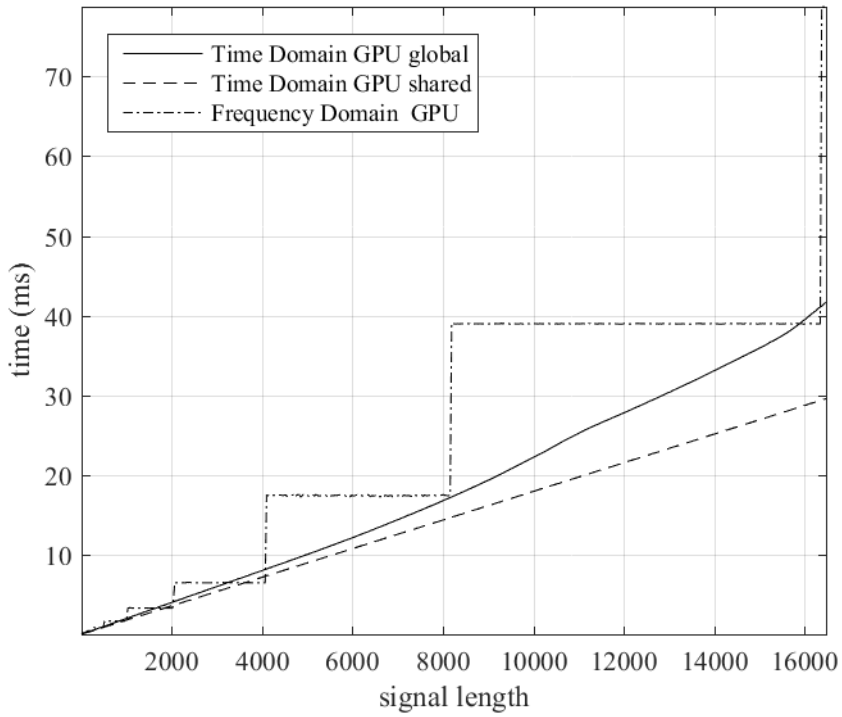


Figure 3.25: Comparison of complex convolution using batch processing on a GPU. The signal length is variable and the filter is fixed at 23 taps.

Table 3.8: Batched convolution execution times with for a 12,672 sample signal and cascaded 23 and 186 tap filter on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
GPU time domain global	ConvGPU	228.8
GPU time domain shared	ConvGPUshared	205.0
GPU frequency domain	cuFFT	39.0

ing batch processing. Cascaded-convolution using batch processing in the frequency domain is fastest.

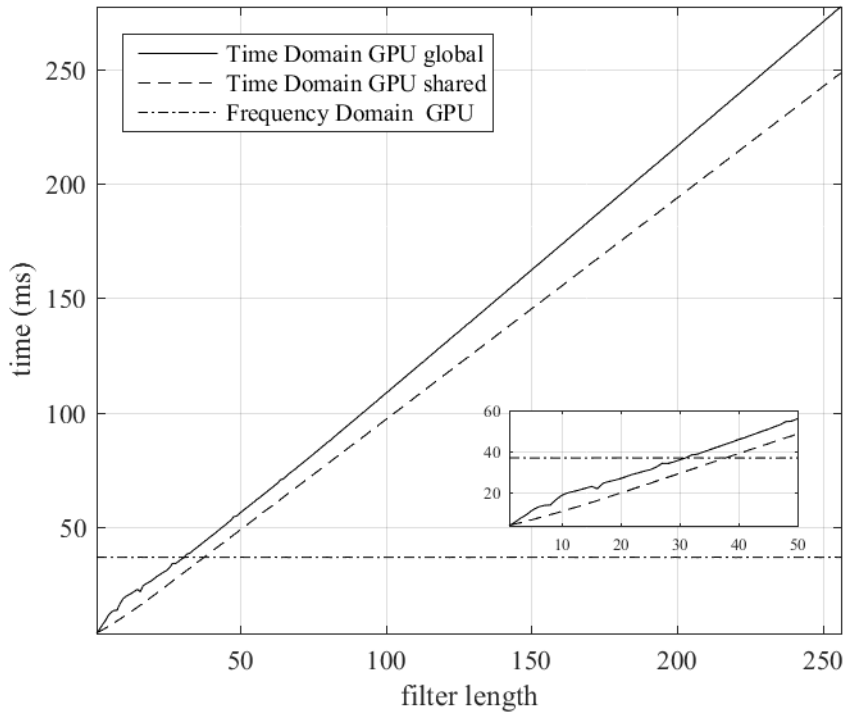


Figure 3.26: Comparison of complex convolution using batch processing on a GPU. The filter length is variable and the signal length is set to 12,672 samples.

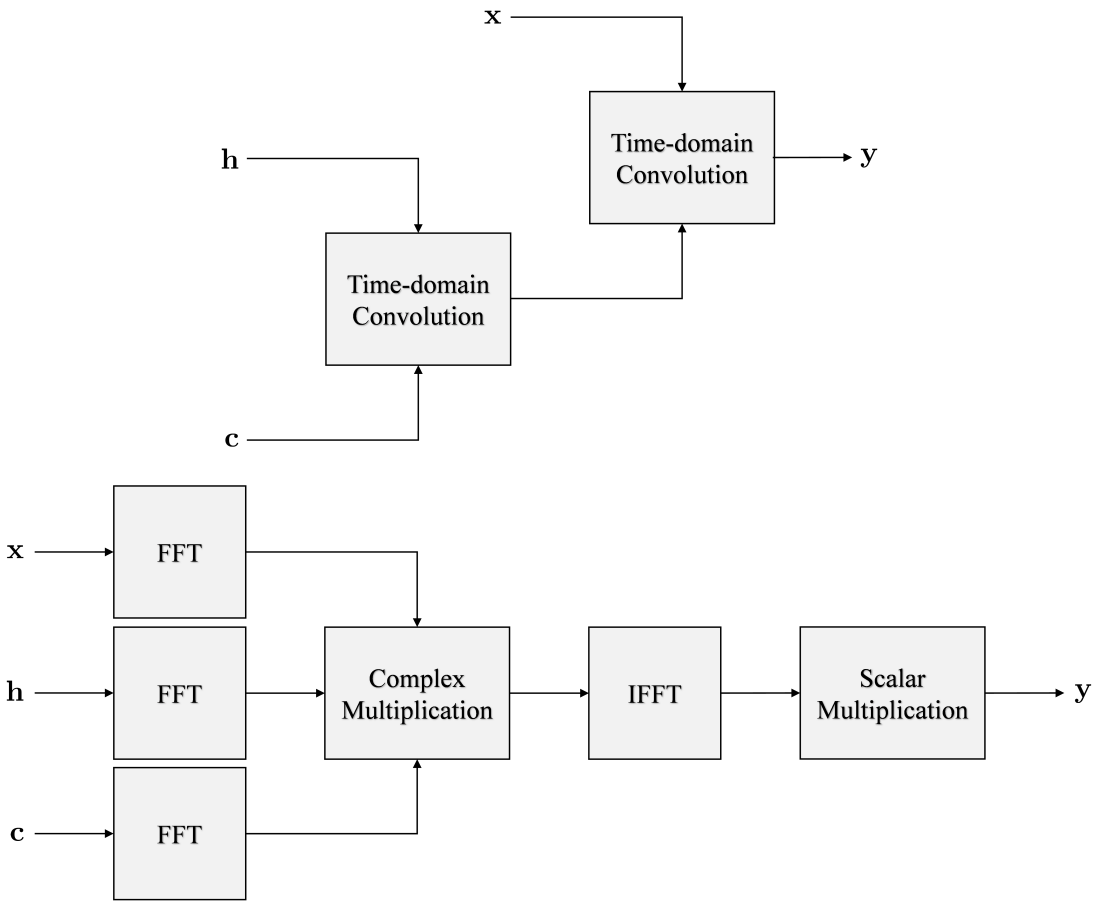


Figure 3.27: Block diagrams showing showing cascaded time-domain convolution and frequency-domain convolution.

Listing 3.5: CUDA code to performing complex convolution five different ways: time domain CPU, frequency domain CPU time domain GPU, time domain GPU using shared memory and frequency domain GPU.

```
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <cufft.h>
#include <fstream>
#include <string>
#include <fftw3.h>
using namespace std;


void ConvCPU(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
        for(int yIdx = 0; yIdx < Lx+Lh-1; yIdx++){
                cufftComplex temp;
                temp.x = 0;
                temp.y = 0;
                for(int hIdx = 0; hIdx < Lh; hIdx++){
                        int xAccessIdx = yIdx-hIdx;
                        if(xAccessIdx>=0 && xAccessIdx<Lx){
                                // temp += x[xAccessIdx]*h[hIdx];
                                float A = x[xAccessIdx].x;
                                float B = x[xAccessIdx].y;
                                float C = h[hIdx].x;
                                float D = h[hIdx].y;
                                cufftComplex result;
                                result.x = A*C-B*D;
                                result.y = A*D+B*C;
                                temp.x += result.x;
                                temp.y += result.y;
                        }
                }
                y[yIdx] = temp;
        }

}

__global__ void ConvGPU(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
        int yIdx = blockIdx.x*blockDim.x + threadIdx.x;

        int lastThread = Lx+Lh-1;

        // don't access elements out of bounds
        if(yIdx >= lastThread)
                return;

        cufftComplex temp;
        temp.x = 0;
        temp.y = 0;
        for(int hIdx = 0; hIdx < Lh; hIdx++){
                int xAccessIdx = yIdx-hIdx;
                if(xAccessIdx>=0 && xAccessIdx<Lx){
                        // temp += x[xAccessIdx]*h[hIdx];
                        float A = x[xAccessIdx].x;
                        float B = x[xAccessIdx].y;
                        float C = h[hIdx].x;
                        float D = h[hIdx].y;
                        cufftComplex result;
                        result.x = A*C-B*D;
                        result.y = A*D+B*C;
                        temp.x += result.x;
                        temp.y += result.y;
                }
        }
        y[yIdx] = temp;
```

```
64  }
65
66
67  __global__ void ConvGPUshared(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh)
        {
68          int yIdx = blockIdx.x*blockDim.x + threadIdx.x;
69
70          int lastThread = Lx+Lh-1;
71
72          extern __shared__ cufftComplex h_shared[];
73          if(threadIdx.x < Lh){
74                  h_shared[threadIdx.x] = h[threadIdx.x];
75          }
76          __syncthreads();
77
78          // don't access elements out of bounds
79          if(yIdx >= lastThread)
80                  return;
81
82          cufftComplex temp;
83          temp.x = 0;
84          temp.y = 0;
85          for(int hIdx = 0; hIdx < Lh; hIdx++){
86                  int xAccessIdx = yIdx-hIdx;
87                  if(xAccessIdx>=0 && xAccessIdx<Lx){
88                          // temp += x[xAccessIdx]*h[hIdx];
89                          float A = x[xAccessIdx].x;
90                          float B = x[xAccessIdx].y;
91                          float C = h_shared[hIdx].x;
92                          float D = h_shared[hIdx].y;
93                          cufftComplex result;
94                          result.x = A*C-B*D;
95                          result.y = A*D+B*C;
96                          temp.x += result.x;
97                          temp.y += result.y;
98                  }
99          }
100         y[yIdx] = temp;
101 }
102
103 __global__ void PointToPointMultiply(cufftComplex* v0, cufftComplex* v1, int lastThread){
104         int i = blockIdx.x*blockDim.x + threadIdx.x;
105
106         // don't access elements out of bounds
107         if(i >= lastThread)
108                 return;
109         float A = v0[i].x;
110         float B = v0[i].y;
111         float C = v1[i].x;
112         float D = v1[i].y;
113
114         // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
115         cufftComplex result;
116         result.x = A*C-B*D;
117         result.y = A*D+B*C;
118
119         v0[i] = result;
120 }
121
122 __global__ void ScalarMultiply(cufftComplex* vec0, float scalar, int lastThread){
123         int i = blockIdx.x*blockDim.x + threadIdx.x;
124
125         // Don't access elements out of bounds
126         if(i >= lastThread)
127                 return;
128         cufftComplex scalarMult;
129         scalarMult.x = vec0[i].x*scalar;
130         scalarMult.y = vec0[i].y*scalar;
```

```
131         vec0[i] = scalarMult;
132 }
133
134 int main(){
135         int N = 1000;
136         int L = 186;
137         int C = N + L - 1;
138         int M = pow(2, ceil(log(C)/log(2)));
139
140         cufftComplex *mySignal1;
141         cufftComplex *mySignal2;
142         cufftComplex *mySignal2_fft;
143
144         cufftComplex *myFilter1;
145         cufftComplex *myFilter2;
146         cufftComplex *myFilter2_fft;
147
148         cufftComplex *myConv1;
149         cufftComplex *myConv2;
150         cufftComplex *myConv2_timeReversed;
151         cufftComplex *myConv3;
152         cufftComplex *myConv4;
153         cufftComplex *myConv5;
154
155         mySignal1               = (cufftComplex*)malloc(N*sizeof(cufftComplex));
156         mySignal2               = (cufftComplex*)malloc(M*sizeof(cufftComplex));
157         mySignal2_fft           = (cufftComplex*)malloc(M*sizeof(cufftComplex));
158
159         myFilter1               = (cufftComplex*)malloc(L*sizeof(cufftComplex));
160         myFilter2               = (cufftComplex*)malloc(M*sizeof(cufftComplex));
161         myFilter2_fft           = (cufftComplex*)malloc(M*sizeof(cufftComplex));
162
163         myConv1                 = (cufftComplex*)malloc(C*sizeof(cufftComplex));
164         myConv2                 = (cufftComplex*)malloc(M*sizeof(cufftComplex));
165         myConv2_timeReversed= (cufftComplex*)malloc(M*sizeof(cufftComplex));
166         myConv3                 = (cufftComplex*)malloc(C*sizeof(cufftComplex));
167         myConv4                 = (cufftComplex*)malloc(C*sizeof(cufftComplex));
168         myConv5                 = (cufftComplex*)malloc(M*sizeof(cufftComplex));
169
170         srand(time(0));
171         for(int i = 0; i < N; i++){
172                 mySignal1[i].x = rand()%100-50;
173                 mySignal1[i].y = rand()%100-50;
174         }
175
176         for(int i = 0; i < L; i++){
177                 myFilter1[i].x = rand()%100-50;
178                 myFilter1[i].y = rand()%100-50;
179         }
180
181         cufftComplex *dev_mySignal3;
182         cufftComplex *dev_mySignal4;
183         cufftComplex *dev_mySignal5;
184
185         cufftComplex *dev_myFilter3;
186         cufftComplex *dev_myFilter4;
187         cufftComplex *dev_myFilter5;
188
189         cufftComplex *dev_myConv3;
190         cufftComplex *dev_myConv4;
191         cufftComplex *dev_myConv5;
192
193         cudaMalloc(&dev_mySignal3, N*sizeof(cufftComplex));
194         cudaMalloc(&dev_mySignal4, N*sizeof(cufftComplex));
195         cudaMalloc(&dev_mySignal5, M*sizeof(cufftComplex));
196
197         cudaMalloc(&dev_myFilter3, L*sizeof(cufftComplex));
198         cudaMalloc(&dev_myFilter4, L*sizeof(cufftComplex));
```

```
199         cudaMalloc(&dev_myFilter5, M*sizeof(cufftComplex));
200
201         cudaMalloc(&dev_myConv3,   C*sizeof(cufftComplex));
202         cudaMalloc(&dev_myConv4,   C*sizeof(cufftComplex));
203         cudaMalloc(&dev_myConv5,   M*sizeof(cufftComplex));
204
205
206         /**
207          * Time-domain Convolution CPU
208          */
209         ConvCPU(myConv1,mySignal1,myFilter1,N,L);
210
211         /**
212          * Frequency Domain Convolution CPU
213          */
214         fftwf_plan forwardPlanSignal = fftwf_plan_dft_1d(M, (fftwf_complex*)mySignal2,    (
                fftwf_complex*)mySignal2_fft,           FFTW_FORWARD, FFTW_MEASURE);
215         fftwf_plan forwardPlanFilter = fftwf_plan_dft_1d(M, (fftwf_complex*)myFilter2,   (
                fftwf_complex*)myFilter2_fft,            FFTW_FORWARD, FFTW_MEASURE);
216         fftwf_plan backwardPlanConv  = fftwf_plan_dft_1d(M, (fftwf_complex*)mySignal2_fft,(
                fftwf_complex*)myConv2_timeReversed, FFTW_FORWARD, FFTW_MEASURE);
217
218         cufftComplex zero; zero.x = 0; zero.y = 0;
219         for(int i = 0; i < M; i++){
220                 if(i<N)
221                         mySignal2[i] = mySignal1[i];
222                 else
223                         mySignal2[i] = zero;
224
225                 if(i<L)
226                         myFilter2[i] = myFilter1[i];
227                 else
228                         myFilter2[i] = zero;
229         }
230
231         fftwf_execute(forwardPlanSignal);
232         fftwf_execute(forwardPlanFilter);
233
234         for (int i = 0; i < M; i++){
235                 // mySignal2_fft = mySignal2_fft*myFilter2_fft;
236                 float A = mySignal2_fft[i].x;
237                 float B = mySignal2_fft[i].y;
238                 float C = myFilter2_fft[i].x;
239                 float D = myFilter2_fft[i].y;
240                 cufftComplex result;
241                 result.x = A*C-B*D;
242                 result.y = A*D+B*C;
243                 mySignal2_fft[i] = result;
244         }
245
246         fftwf_execute(backwardPlanConv);
247
248         // myConv2 from fftwf must be time reversed and scaled
249         // to match Matlab, myConv1, myConv3, myConv4 and myConv5
250         cufftComplex result;
251         for (int i = 0; i < M; i++){
252                 result.x = myConv2_timeReversed[M-i].x/M;
253                 result.y = myConv2_timeReversed[M-i].y/M;
254                 myConv2[i] = result;
255         }
256         result.x = myConv2_timeReversed[0].x/M;
257         result.y = myConv2_timeReversed[0].y/M;
258         myConv2[0] = result;
259
260         fftwf_destroy_plan(forwardPlanSignal);
261         fftwf_destroy_plan(forwardPlanFilter);
262         fftwf_destroy_plan(backwardPlanConv);
263
```

```
264
265         /**
266          * Time-domain Convolution GPU Using Global Memory
267          */
268         cudaMemcpy(dev_mySignal3, mySignal1, sizeof(cufftComplex)*N, cudaMemcpyHostToDevice)
                ;
269         cudaMemcpy(dev_myFilter3, myFilter1, sizeof(cufftComplex)*L, cudaMemcpyHostToDevice)
                ;
270
271         int T_B = 512;
272         int B = C/T_B;
273         if(C % T_B > 0)
274                 B++;
275         ConvGPU<<<B, T_B>>>(dev_myConv3, dev_mySignal3, dev_myFilter3, N, L);
276
277         cudaMemcpy(myConv3, dev_myConv3, C*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
278
279
280         /**
281          * Time-domain Convolution GPU Using Shared Memory
282          */
283         cudaMemcpy(dev_mySignal4, mySignal1, sizeof(cufftComplex)*N, cudaMemcpyHostToDevice)
                ;
284         cudaMemcpy(dev_myFilter4, myFilter1, sizeof(cufftComplex)*L, cudaMemcpyHostToDevice)
                ;
285
286         T_B = 512;
287         B = C/T_B;
288         if(C % T_B > 0)
289                 B++;
290         ConvGPUshared<<<B, T_B,L*sizeof(cufftComplex)>>>(dev_myConv4, dev_mySignal4,
                dev_myFilter4, N, L);
291
292         cudaMemcpy(myConv4, dev_myConv4, C*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
293
294
295         /**
296          * Frequency-domain Convolution GPU
297          */
298         cufftHandle plan;
299         int n[1] = {M};
300         cufftPlanMany(&plan,1,n,NULL,1,1,NULL,1,1,CUFFT_C2C,1);
301
302         cudaMemset(dev_mySignal5, 0,         M*sizeof(cufftComplex));
303         cudaMemset(dev_myFilter5, 0,         M*sizeof(cufftComplex));
304
305         cudaMemcpy(dev_mySignal5, mySignal2, M*sizeof(cufftComplex), cudaMemcpyHostToDevice)
                ;
306         cudaMemcpy(dev_myFilter5, myFilter2, M*sizeof(cufftComplex), cudaMemcpyHostToDevice)
                ;
307
308         cufftExecC2C(plan, dev_mySignal5, dev_mySignal5, CUFFT_FORWARD);
309         cufftExecC2C(plan, dev_myFilter5, dev_myFilter5, CUFFT_FORWARD);
310
311         T_B = 512;
312         B = M/T_B;
313         if(M % T_B > 0)
314                 B++;
315         PointToPointMultiply<<<B, T_B>>>(dev_mySignal5, dev_myFilter5, M);
316
317         cufftExecC2C(plan, dev_mySignal5, dev_mySignal5, CUFFT_INVERSE);
318
319         T_B = 128;
320         B = M/T_B;
321         if(M % T_B > 0)
322                 B++;
323         float scalar = 1.0/((float)M);
324         ScalarMultiply<<<B, T_B>>>(dev_mySignal5, scalar, M);
```

```
325
326         cudaMemcpy(myConv5, dev_mySignal5, M*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
327
328         cufftDestroy(plan);
329
330         free(mySignal1);
331         free(mySignal2);
332
333         free(myFilter1);
334         free(myFilter2);
335
336         free(myConv1);
337         free(myConv2);
338         free(myConv2_timeReversed);
339         free(myConv3);
340         free(myConv4);
341         free(myConv5);
342         fftwf_cleanup();
343
344         cudaFree(dev_mySignal3);
345         cudaFree(dev_mySignal4);
346         cudaFree(dev_mySignal5);
347
348         cudaFree(dev_myFilter3);
349         cudaFree(dev_myFilter4);
350         cudaFree(dev_myFilter5);
351
352         cudaFree(dev_myConv3);
353         cudaFree(dev_myConv4);
354         cudaFree(dev_myConv5);
355
356         return 0;
357 }
```

Listing 3.6: CUDA code to perform batched complex convolution three different ways in a GPU: time domain using global memory, time domain using shared memory and frequency domain GPU.

```
#include <cufft.h>
#include <iostream>
using namespace std;

__global__ void ConvGPU(cufftComplex* y_out,cufftComplex* x_in,cufftComplex* h_in,int Lx,int
     Lh,int maxThreads){
        int threadNum = blockIdx.x*blockDim.x + threadIdx.x;
        int convLength = Lx+Lh-1;

        // Don't access elements out of bounds
        if(threadNum >= maxThreads)
                return;

        int batch = threadNum/convLength;
        int yIdx  = threadNum%convLength;
        cufftComplex* x = &x_in[Lx*batch];
        cufftComplex* h = &h_in[Lh*batch];
        cufftComplex* y = &y_out[convLength*batch];

        cufftComplex temp;
        temp.x = 0;
        temp.y = 0;
        for(int hIdx = 0; hIdx < Lh; hIdx++){
                int xAccessIdx = yIdx-hIdx;
                if(xAccessIdx>=0 && xAccessIdx<Lx){
                        // temp += x[xAccessIdx]*h[hIdx];
                        // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
                        float A = x[xAccessIdx].x;
                        float B = x[xAccessIdx].y;
                        float C = h[hIdx].x;
                        float D = h[hIdx].y;
                        cufftComplex complexMult;
                        complexMult.x = A*C-B*D;
                        complexMult.y = A*D+B*C;

                        temp.x += complexMult.x;
                        temp.y += complexMult.y;
                }
        }
        y[yIdx] = temp;
}

__global__ void ConvGPUshared(cufftComplex* y_out,cufftComplex* x_in,cufftComplex* h_in,int
    Lx,int Lh,int maxThreads){

        int threadNum = blockIdx.x*blockDim.x + threadIdx.x;
        int convLength = Lx+Lh-1;
        // Don't access elements out of bounds
        if(threadNum >= maxThreads)
                return;

        int batch = threadNum/convLength;
        int yIdx  = threadNum%convLength;
        cufftComplex* x = &x_in[Lx*batch];
        cufftComplex* h = &h_in[Lh*batch];
        cufftComplex* y = &y_out[convLength*batch];

        extern __shared__ cufftComplex h_shared[];
        if(threadIdx.x < Lh)
                h_shared[threadIdx.x] = h[threadIdx.x];

        __syncthreads();

```

```
62          cufftComplex temp;
63          temp.x = 0;
64          temp.y = 0;
65          for(int hIdx = 0; hIdx < Lh; hIdx++){
66                  int xAccessIdx = yIdx-hIdx;
67                  if(xAccessIdx>=0 && xAccessIdx<Lx){
68                          // temp += x[xAccessIdx]*h[hIdx];
69                          // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
70                          float A = x[xAccessIdx].x;
71                          float B = x[xAccessIdx].y;
72                          float C = h_shared[hIdx].x;
73                          float D = h_shared[hIdx].y;
74                          cufftComplex complexMult;
75                          complexMult.x = A*C-B*D;
76                          complexMult.y = A*D+B*C;
77
78                          temp.x += complexMult.x;
79                          temp.y += complexMult.y;
80                  }
81          }
82          y[yIdx] = temp;
83  }
84
85  __global__ void PointToPointMultiply(cufftComplex* vec0, cufftComplex* vec1, int maxThreads)
        {
86          int i = blockIdx.x*blockDim.x + threadIdx.x;
87          // Don't access elements out of bounds
88          if(i >= maxThreads)
89                  return;
90          // vec0[i] = vec0[i]*vec1[i];
91          // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
92          float A = vec0[i].x;
93          float B = vec0[i].y;
94          float C = vec1[i].x;
95          float D = vec1[i].y;
96          cufftComplex complexMult;
97          complexMult.x = A*C-B*D;
98          complexMult.y = A*D+B*C;
99          vec0[i] = complexMult;
100 }
101
102 __global__ void ScalarMultiply(cufftComplex* vec0, float scalar, int lastThread){
103         int i = blockIdx.x*blockDim.x + threadIdx.x;
104         // Don't access elements out of bounds
105         if(i >= lastThread)
106                 return;
107         cufftComplex scalarMult;
108         scalarMult.x = vec0[i].x*scalar;
109         scalarMult.y = vec0[i].y*scalar;
110         vec0[i] = scalarMult;
111 }
112
113 int main(){
114         int numBatches = 3104;
115         int N = 12672;
116         int L = 186;
117         int C = N + L - 1;
118         int M = pow(2, ceil(log(C)/log(2)));
119         int maxThreads;
120         int T_B;
121         int B;
122
123         cufftHandle plan;
124         int n[1] = {M};
125         cufftPlanMany(&plan,1,n,NULL,1,1,NULL,1,1,CUFFT_C2C,numBatches);
126
127         // Allocate memory on host
128         cufftComplex *mySignal1;
```

```
129         cufftComplex *mySignal1_pad;
130         cufftComplex *myFilter1;
131         cufftComplex *myFilter1_pad;
132         cufftComplex *myConv1;
133         cufftComplex *myConv2;
134         cufftComplex *myConv3;
135         mySignal1      = (cufftComplex*) malloc(N*numBatches*sizeof(cufftComplex));
136         mySignal1_pad  = (cufftComplex*) malloc(M*numBatches*sizeof(cufftComplex));
137         myFilter1      = (cufftComplex*) malloc(L*numBatches*sizeof(cufftComplex));
138         myFilter1_pad  = (cufftComplex*) malloc(M*numBatches*sizeof(cufftComplex));
139         myConv1        = (cufftComplex*) malloc(C*numBatches*sizeof(cufftComplex));
140         myConv2        = (cufftComplex*) malloc(C*numBatches*sizeof(cufftComplex));
141         myConv3        = (cufftComplex*) malloc(M*numBatches*sizeof(cufftComplex));
142
143         srand(time(0));
144         for(int i = 0; i < N; i++){
145                 mySignal1[i].x = rand()%100-50;
146                 mySignal1[i].y = rand()%100-50;
147         }
148
149         for(int i = 0; i < L; i++){
150                 myFilter1[i].x = rand()%100-50;
151                 myFilter1[i].y = rand()%100-50;
152         }
153
154         cufftComplex zero;
155         zero.x = 0;
156         zero.y = 0;
157         for(int i = 0; i<M*numBatches; i++){
158                 mySignal1_pad[i] = zero;
159                 myFilter1_pad[i] = zero;
160         }
161         for(int batch=0; batch < numBatches; batch++){
162                 for(int i = 0; i < N; i++){
163                         mySignal1[batch*N+i] = mySignal1[i];
164                         mySignal1_pad[batch*M+i] = mySignal1[i];
165                 }
166                 for(int i = 0; i < L; i++){
167                         myFilter1[batch*L+i] = myFilter1[i];
168                         myFilter1_pad[batch*M+i] = myFilter1[i];
169                 }
170         }
171
172         // Allocate memory on device
173         cufftComplex *dev_mySignal1;
174         cufftComplex *dev_mySignal2;
175         cufftComplex *dev_mySignal3;
176         cufftComplex *dev_myFilter1;
177         cufftComplex *dev_myFilter2;
178         cufftComplex *dev_myFilter3;
179         cufftComplex *dev_myConv1;
180         cufftComplex *dev_myConv2;
181         cufftComplex *dev_myConv3;
182         cudaMalloc(&dev_mySignal1, N*numBatches*sizeof(cufftComplex));
183         cudaMalloc(&dev_mySignal2, N*numBatches*sizeof(cufftComplex));
184         cudaMalloc(&dev_mySignal3, M*numBatches*sizeof(cufftComplex));
185         cudaMalloc(&dev_myFilter1, L*numBatches*sizeof(cufftComplex));
186         cudaMalloc(&dev_myFilter2, L*numBatches*sizeof(cufftComplex));
187         cudaMalloc(&dev_myFilter3, M*numBatches*sizeof(cufftComplex));
188         cudaMalloc(&dev_myConv1,   C*numBatches*sizeof(cufftComplex));
189         cudaMalloc(&dev_myConv2,   C*numBatches*sizeof(cufftComplex));
190         cudaMalloc(&dev_myConv3,   M*numBatches*sizeof(cufftComplex));
191
192         /**
193          * Time-domain Convolution GPU Using Global Memory
194          */
195         cudaMemcpy(dev_mySignal1, mySignal1, numBatches*sizeof(cufftComplex)*N,
              cudaMemcpyHostToDevice);
```

```
196         cudaMemcpy(dev_myFilter1, myFilter1, numBatches*sizeof(cufftComplex)*L,
                cudaMemcpyHostToDevice);
197
198         maxThreads = C*numBatches;
199         T_B = 128;
200         B = maxThreads/T_B;
201         if(maxThreads % T_B > 0)
202                 B++;
203         ConvGPU<<<B, T_B>>>(dev_myConv1, dev_mySignal1, dev_myFilter1, N, L, maxThreads);
204
205         cudaMemcpy(myConv1, dev_myConv1, C*numBatches*sizeof(cufftComplex),
                cudaMemcpyDeviceToHost);
206
207         /**
208          * Time-domain Convolution GPU Using Shared Memory
209          */
210         cudaMemcpy(dev_mySignal2, mySignal1, numBatches*sizeof(cufftComplex)*N,
                cudaMemcpyHostToDevice);
211         cudaMemcpy(dev_myFilter2, myFilter1, numBatches*sizeof(cufftComplex)*L,
                cudaMemcpyHostToDevice);
212
213         maxThreads = C*numBatches;
214         T_B = 256;
215         B = maxThreads/T_B;
216         if(maxThreads % T_B > 0)
217                 B++;
218         ConvGPUshared<<<B, T_B, L*sizeof(cufftComplex)>>>(dev_myConv2, dev_mySignal2,
                dev_myFilter2, N, L,maxThreads);
219
220         cudaMemcpy(myConv2, dev_myConv2, C*numBatches*sizeof(cufftComplex),
                cudaMemcpyDeviceToHost);
221
222         /**
223          * Frequency-domain Convolution GPU
224          */
225         cudaMemcpy(dev_mySignal3, mySignal1_pad, M*numBatches*sizeof(cufftComplex),
                cudaMemcpyHostToDevice);
226         cudaMemcpy(dev_myFilter3, myFilter1_pad, M*numBatches*sizeof(cufftComplex),
                cudaMemcpyHostToDevice);
227
228         cufftExecC2C(plan, dev_mySignal3, dev_mySignal3, CUFFT_FORWARD);
229         cufftExecC2C(plan, dev_myFilter3, dev_myFilter3, CUFFT_FORWARD);
230
231         maxThreads = M*numBatches;
232         T_B = 96;
233         B = maxThreads/T_B;
234         if(maxThreads % T_B > 0)
235                 B++;
236         PointToPointMultiply<<<B, T_B>>>(dev_mySignal3, dev_myFilter3, maxThreads);
237         cufftExecC2C(plan, dev_mySignal3, dev_mySignal3, CUFFT_INVERSE);
238
239         T_B = 640;
240         B = maxThreads/T_B;
241         if(maxThreads % T_B > 0)
242                 B++;
243         float scalar = 1.0/((float)M);
244         ScalarMultiply<<<B, T_B>>>(dev_mySignal3, scalar, maxThreads);
245
246         cudaMemcpy(myConv3, dev_mySignal3, M*numBatches*sizeof(cufftComplex),
                cudaMemcpyDeviceToHost);
247
248         cufftDestroy(plan);
249
250         // Free vectors on CPU
251         free(mySignal1);
252         free(myFilter1);
253         free(myConv1);
254         free(myConv2);
```

```
255         free(myConv3);
256
257         // Free vectors on GPU
258         cudaFree(dev_mySignal1);
259         cudaFree(dev_mySignal2);
260         cudaFree(dev_mySignal3);
261         cudaFree(dev_myFilter1);
262         cudaFree(dev_myFilter2);
263         cudaFree(dev_myFilter3);
264         cudaFree(dev_myConv1);
265         cudaFree(dev_myConv2);
266         cudaFree(dev_myConv3);
267
268         return 0;
269 }
```

CHAPTER 4. EQUALIZER GPU IMPLEMENTATION AND BIT ERROR RATE PERFORMANCE

4.1 GPU Implementation

Each equalizer in the PAQ system presents an interesting challenge from a GPU implementation perspective. The equations for each equalizer were presented in Section 2.3.5. In this chapter, the equalizer equations are reformulated for fast and efficient GPU implementation,

Every equalizer filter is computed using batch processing. In batch processing, each packet is independent of all other packets. Each packet in a batch is processed in exactly the same way with different data. To simplify the figures, every block diagram in this chapter shows how one packet is processed. The processing is repeated 3104 times to compute a full batch of equalizer filters.

Convolution is used many times in this chapter. Section 3.2.3 showed that GPU frequency-domain convolution using batch processing performs best for the PAQ system. To simplify block diagrams, frequency-domain convolution is shown as one block, as illustrated in Figures 4.1 and 4.2.

Note that the detection filter $\mathbf{d}$ and the SOQPSK-TG power spectral density $\blacksquare$ are defined constants. The SOQPSK-TG power spectral density $\blacksquare$ and $\mathbf{D}$, the 16,384-point FFT of $\mathbf{d}$, are precomputed and stored. Applying $\mathbf{d}$ in the frequency domain does not require an extra FFT, only extra complex multiplies.

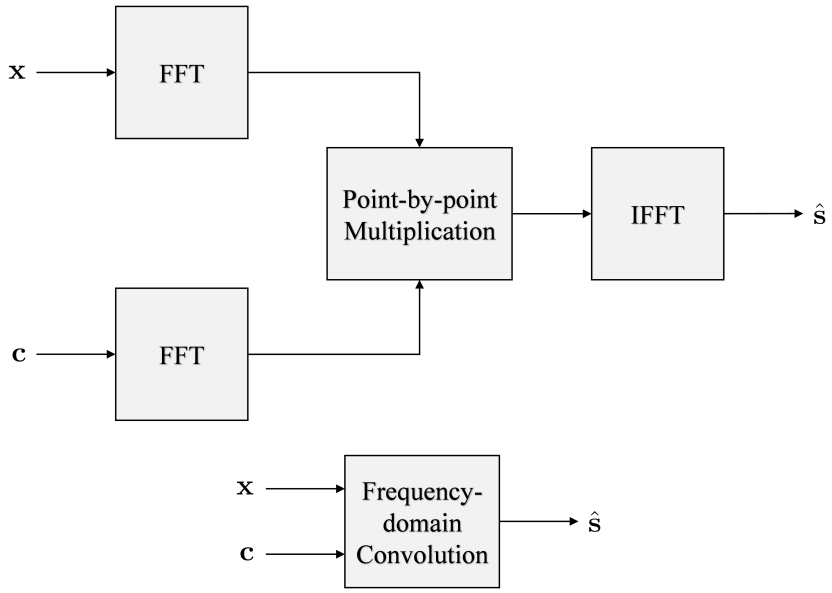


Figure 4.1: A block diagram representation of $\mathbf{y} = \mathbf{x} * \mathbf{c}$. Convolution is performed in the frequency domain. All the required operations in the top part of the figure are represented by the block in the lower part of the figure.

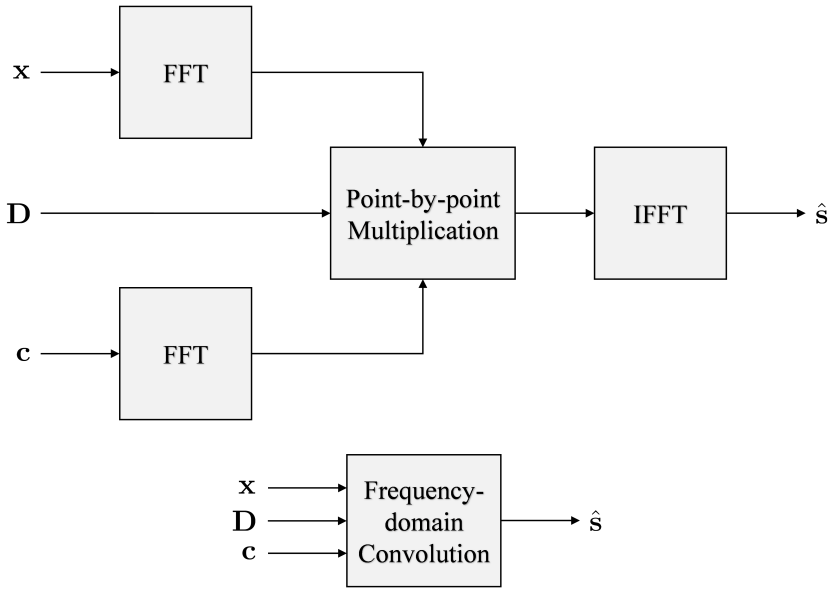


Figure 4.2: A block diagram representation of $\mathbf{y} = (\mathbf{x} * \mathbf{c}) * \mathbf{D}$ (cascaded convolution), where $\mathbf{D}$ is the FFT of a filter that does not change with the data (i.e. $\mathbf{D}$ is precomputed and stored. Convolution is performed in the frequency domain. All the required operations in the top part of the figure are represented by the block in the lower part of the figure.

4.1.1 Zero-forcing and MMSE GPU Implementation

The ZF and MMSE FIR equalizer filter coefficient computations have exactly the same form as shown in Equations (2.17) and (2.22). Consequently any algorithm reconstructing that improves the efficiency of computing the MMSE equalizer coefficients, also applies to computing the ZF filter coefficients. Three approaches to computing the equalizer filter coefficients were explored:

- Using the Levinson-Durbin recursion algorithm to solve $\mathbf{R}\mathbf{c}_{\text{MMSE}} = \hat{\mathbf{g}}$ leveraging the toeplitz property of $\mathbf{R}$.
- Using the cuBLAS LU decomposition library to compute the inverse and matrix vector multiplication defined by $\mathbf{c}_{\text{MMSE}} = \mathbf{R}^{-1}\hat{\mathbf{g}}$.
- Using the cuSolver library to solve $\mathbf{R}\mathbf{c}_{\text{MMSE}} = \hat{\mathbf{g}}$ by leveraging the sparse property of $\mathbf{R}$.

For reference, the matrix $\mathbf{R}$ is

$$\mathbf{R} = \begin{bmatrix} r_{\hat{h}}(0) + \hat{\sigma}_w^2 & r_{\hat{h}}^*(1) & \cdots & r_{\hat{h}}^*(L_{eq}-1) \\ r_{\hat{h}}(1) & r_{\hat{h}}(0) + \hat{\sigma}_w^2 & \cdots & r_{\hat{h}}^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_{\hat{h}}(L_{eq}-1) & r_{\hat{h}}(L_{eq}-2) & \cdots & r_{\hat{h}}(0) + \hat{\sigma}_w^2 \end{bmatrix}. \tag{4.1}$$

The Levinson-Durbin recursion algorithm avoids the $\mathcal{O}(n^3)$ operations normally associated with solvers by leveraging the Toeplitz or diagonal-constant structure of $\mathbf{R}$ [19, Chap. 5]. The first GPU implementation of the Levinson-Durbin recursion algorithm computed the MMSE equalizer filter assuming the matrix $\mathbf{R}$ and the vector $\hat{\mathbf{g}}$ were real-valued. The Levinson-Durbin recursion algorithm showed promise by computing 3104 real-valued MMSE equalizer filters in 500 ms. The GPU implementation of Levinson-Durbin recursion was then converted to the complex-valued case. The Levinson-Durbin recursion computed 3104 complex-valued MMSE equalizer filters in 2,500 ms, in excess of the 1907 ms maximum for all processing time.

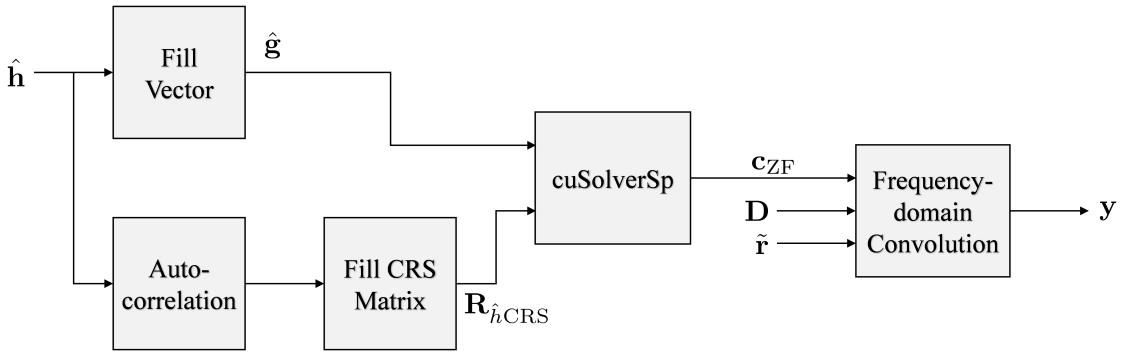


Figure 4.3: Block diagram showing how the zero-forcing equalizer coefficients are implemented in the GPU.

The next algorithm explored computed the inverse of $\mathbf{R}$ using the cuBLAS batch processing library. The cuBLAS library computes a *complex-valued* inverse using the LU decomposition in 600 ms. cuBLAS executed faster than the Levinson-Durbin recursion algorithm, but 600 ms is still 31% of the total 1907 ms processing time.

The final and fastest algorithm explored solves $\mathbf{c}_{\text{MMSE}} = \mathbf{R}^{-1}\hat{\mathbf{g}}$ by leveraging the sparse properties of $\mathbf{R}$ using the cuSolverSp batch processing library. The 186×186) auto-correlation matrix $\mathbf{R}$ comprises the sample auto-correlation $r_{\hat{h}}(k)$ and the noise variance estimate $\hat{\sigma}_w^2$. Because the sample auto-correlation $r_{\hat{h}}(k)$ only has support on $-37 \leq k \leq 37$ and the addition of $\hat{\sigma}_w^2$ does not increase that support, the auto-correlation matrix $\mathbf{R}$ is sparse: 63% of its entries are zeroes. "cusolverSpCcsrqrsvBatched" is the GPU function used from the cuSolverSp library. cusolverSpCcsrqrsvBatched is a complex-valued solver that leverages batch processing and the sparse properties of $\mathbf{R}$ by exploiting Compressed Row Storage (CRS) [20]. CRS reduces the large 186×186 matrix to a 12544 element CSR matrix $\mathbf{R}_{\text{CRS}}$. Before cusolverSpCcsrqrsvBatched can be called, the CSR matrix $\mathbf{R}_{\text{CRS}}$ has to be built and the vector $\hat{\mathbf{g}}$ has to be built. An example of how to use the CUDA cusolverSp library can be found in [21].

Figures 4.3 and 4.4 show how the ZF and MMSE equalizer filters are computed and applied to the received samples, respectively. Note that the equalizer filters are applied in the frequency-domain with the detection filter. Table 4.1 lists the algorithms researched and their respective execution times.

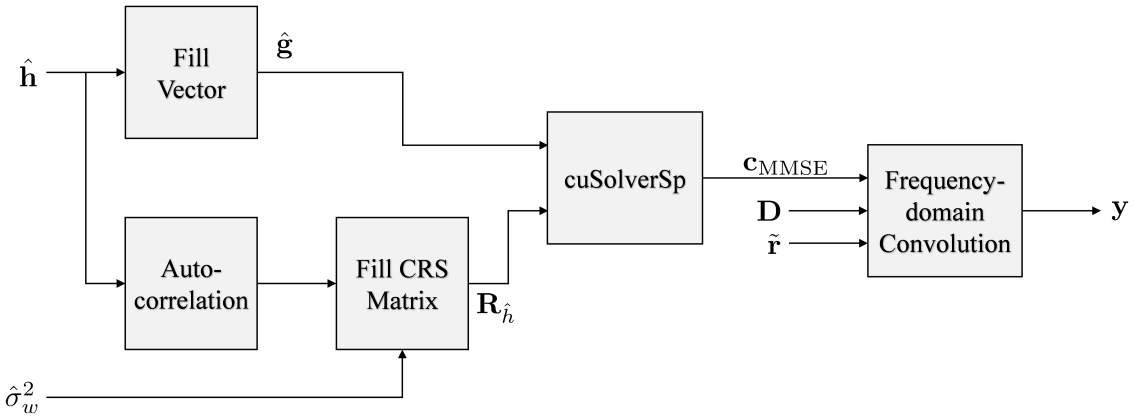


Figure 4.4: Block diagram showing how the minimum mean-squared error equalizer coefficients are implemented in the GPU.

Table 4.1: Algorithms used to compute the ZF and MMSE equalizer filters.

Algorithm	Data type	Execution Time (ms)
Levinson Recursion	Real	500
Levinson Recursion	Complex	2500
LU Decomposition	Complex	600
cuSolver	Complex	356

4.1.2 Constant Modulus Algorithm GPU Implementation

The CMA equalizer is an adaptive FIR filter, where the filter coefficients are updated on a packet-by-packet basis using a steepest descent algorithm shown in Equation (2.26). The more iterations the steepest descent algorithm executes, the more effective the CMA equalizer is. The most computationally heavy operation in the CMA update is computing ∇J, shown in Equation (2.27) and repeated here for reference

$$\nabla J = \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \left[\hat{s}^{(b)}(n) \left(\hat{s}^{(b)}(n) \right)^* - 1 \right] \hat{s}^{(b)}(n) \tilde{\mathbf{r}}^*(n), \tag{4.2}$$

where $\tilde{\mathbf{r}}^*(n)$ is defined in (2.28). Two approaches to computing the equalizer filter coefficients were explored:

- Computing ∇J directly using the summation in Equation (4.2).
- Reformulating ∇J into convolution to leverage the fast computation time of convolution using batch processing in GPUs.

To simplify the analysis of each option, (4.2) is expressed as

$$\nabla J = \frac{1}{L_{\text{pkt}}} \sum_{n=0}^{L_{\text{pkt}}-1} z(n)\tilde{\mathbf{r}}^*(n), \tag{4.3}$$

where

$$z(n) = 2\left[\hat{s}^{(b)}(n)(\hat{s}^{(b)}(n))^* - 1\right]\hat{s}^{(b)}(n). \tag{4.4}$$

The first (direct) approach required 421.3 ms of one summation. This approach did not allow for multiple iterations. The poor performance is due to the fact that the GPU kernel is memory bandwidth limited.

To avoid the problems with the first approach, the second approach refoumulated the summation (4.3) as a convolution. The reformulations allows the GPUs to leverage the computational efficiencies of the convolution outlined in Chapter 3.2. To begin, the expression of ∇J is expanded as follows:

$$\nabla J = \frac{z(0)}{L_{\text{pkt}}}\begin{bmatrix} \tilde{r}^*(L_1) \\ \vdots \\ \tilde{r}^*(0) \\ \vdots \\ \tilde{r}^*(L_2) \end{bmatrix} + \frac{z(1)}{L_{\text{pkt}}}\begin{bmatrix} \tilde{r}^*(1+L_1) \\ \vdots \\ \tilde{r}^*(1) \\ \vdots \\ \tilde{r}^*(1-L_2) \end{bmatrix} + \cdots \frac{z(L_{\text{pkt}}-1)}{L_{\text{pkt}}}\begin{bmatrix} \tilde{r}^*(L_{\text{pkt}}-1+L_1) \\ \vdots \\ \tilde{r}^*(L_{\text{pkt}}-1) \\ \vdots \\ \tilde{r}^*(L_{\text{pkt}}-1-L_2) \end{bmatrix}. \tag{4.5}$$

This reveals a pattern in the relationship between $z(n)$ and $\mathbf{r}(n)$. The kth value of ∇J is

$$\nabla J(k) = \frac{1}{L_{\text{pkt}}} \sum_{m=0}^{L_{\text{pkt}}-1} z(m)\tilde{r}^*(m-k), \quad -L_1 \leq k \leq L_2. \tag{4.6}$$

The summation almost looks like a convolution accept the conjugate on the element $\tilde{r}(n)$. To put the summation into the familiar convolution form, define

$$\rho(n) = \tilde{r}^*(-n). \tag{4.7}$$

Now

$$\nabla J(k) = \frac{1}{L_{\text{pkt}}} \sum_{m=0}^{L_{\text{pkt}}-1} z(m)\rho(k-m). \tag{4.8}$$

Note that $z(n)$ has support on $0 \leq n \leq L_{\text{pkt}} - 1$ and $\rho(n)$ has support on $-L_{\text{pkt}} + 1 \leq n \leq 0$, the result of the convolution sum $\gamma(n)$ has support on $-L_{\text{pkt}} + 1 \leq n \leq L_{\text{pkt}} - 1$. Putting all the pieces together, we have

$$\begin{aligned} \gamma(n) &= \sum_{m=0}^{L_{\text{pkt}}-1} z(m)\rho(n-m) \\ &= \sum_{m=0}^{L_{\text{pkt}}-1} z(m)\tilde{r}^*(m-n), \end{aligned} \tag{4.9}$$

Comparing Equation (4.8) and (4.9) shows that

$$\nabla J(k) = \frac{1}{L_{\text{pkt}}}\gamma(-k), \quad -L_1 \leq k \leq L_2. \tag{4.10}$$

The values of interest are shown in Figure 4.5. This suggests the matlab code shown in Table 4.2 to compute the gradient vector ∇J and implementation of the CMA equalizer.

Using convolution to compute ∇J decreased execution time significantly. 88.8 ms is required for one CMA iteration. Note that all other frequency-domain convolutions in this thesis are computed using 16,384-point FFTs. A length 32,768 FFT is required because the result of the convolution $z(n) * \rho(n)$ is $2L_{\text{pkt}} - 1 = 25{,}343$.

Figure 4.6 shows a block diagram of how the CMA equalizer runs on the GPU. Note that the detection filter is applied only on the last iteration. Table 4.3 lists the comparison on computing $\nabla J(k)$ using convolution. By reformulating the computation of ∇J, the execution time was reduced

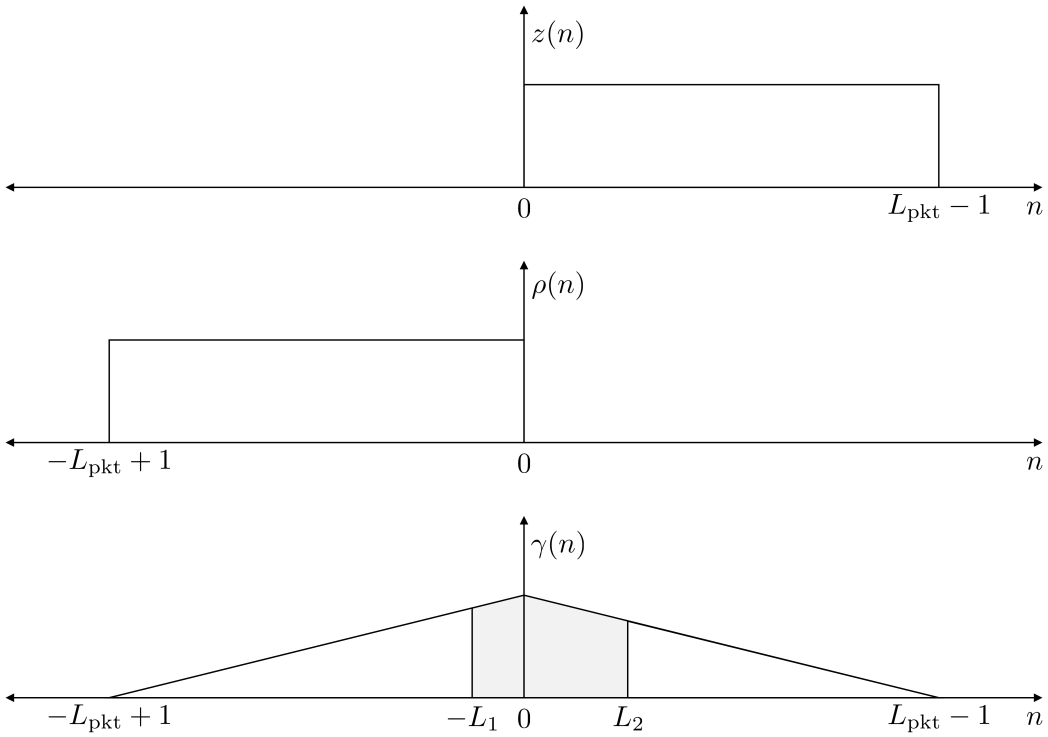


Figure 4.5: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $\gamma(n)$.

Table 4.2: MATLAB code listing for the CMA equalizer.

```
 1 c_CMA = c_MMSE;
 2 for i = 1:its
 3     ss = conv(r,c_CMA);
 4     s = ss(L1+1:end-L2); % trim s
 5     z = 2*(y.*conj(y)-1).*s;
 6     Z = fft(z,Nfft);
 7     RT = fft(conj(rt(end:-1:1)),Nfft)
 8     gamma = ifft(Z.*RT);
 9     delJ = gamma(Lpkt-L1:Lpkt+L2)/Lpkt;
10     c_CMA = c_CMA-mu*delJ;
11 end
12 yy = conv(r,c_CMA);
13 y = yy(L1+1:end-L2); % trim yy
```

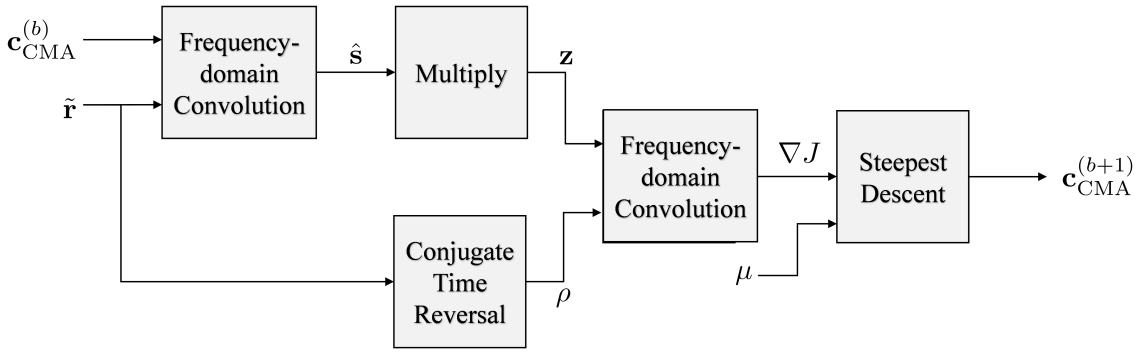


Figure 4.6: Block diagram showing how the CMA equalizer filter is implemented in the GPU using frequency-domain convolution twice per iteration.

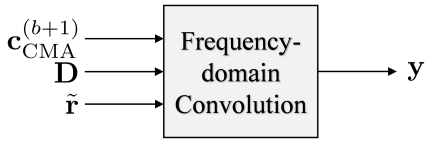


Figure 4.7: After the final CMA iteration, the de-rotated samples are filtered by the detection filter and the CMA equalizer in the frequency domain.

by a factor of 4.74. Implementing ∇J directly only provided time for 2 iterations, while using convolution to compute $\nabla J(k)$ provided time for 12 iterations.

4.1.3 Frequency Domain Equalizer GPU Implementations

The FFT-domain transfer function for FDE1 is given by (2.30). The FFT of the equalizer output is simply the product of the point-by-point multiplication involving (2.30) and the length-N_{FFT} of the samples in $\tilde{\mathbf{r}}$, denoted $\tilde{R}(e^{j\omega_k})$ for $k = 0, 1, \ldots, N_{\mathrm{FFT}}$. Consequently, the FFT of the

Table 4.3: Algorithms used to compute the cost function gradient ∇J.

CMA Iteration Algorithm	Execution Time (ms)
∇J directly	421.317
∇J using convolution	88.774

Table 4.4: Execution times for calculating and applying Frequency Domain Equalizer One and Two.

Algorithm	Execution Time (ms)
Frequency Domain Equalizer One	57.156
Frequency Domain Equalizer Two	58.841

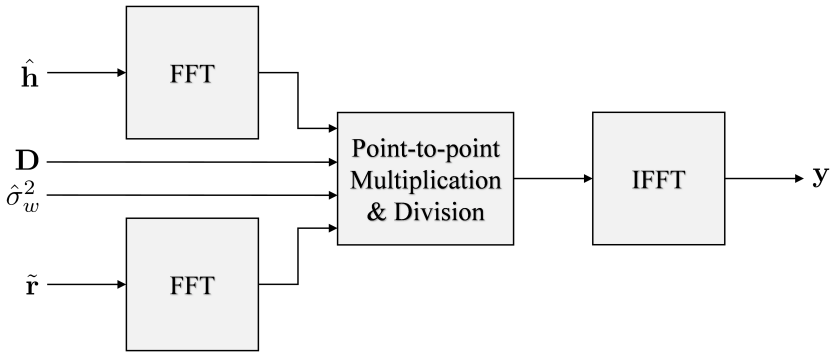


Figure 4.8: Block diagram showing frequency domain equalizer one is implemented in the frequency domain in GPUs.

equalizer output is

$$\hat{S}_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})\tilde{R}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{\text{FFT}}} \text{ for } k = 0, 1, \cdots, N_{\text{FFT}} - 1. \tag{4.11}$$

Applying the detection filter produces

$$Y_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})\tilde{R}(e^{j\omega_k})D(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{\text{FFT}}} \text{ for } k = 0, 1, \cdots, N_{\text{FFT}} - 1. \tag{4.12}$$

The computations for FDE2 are identical to (4.11) and (4.12), except for the inclusion of $\Psi(e^{j\omega_k})$ [cf, (2.31)]. Figures 4.8 and 4.9 show the block diagrams for GPU implementation of FDE1 and FDE2. Table 4.4 shows the execution times for calculating and applying FDE1 and FDE2.

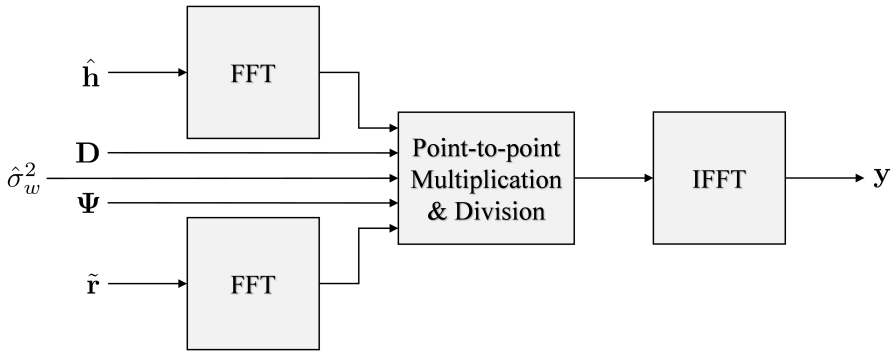


Figure 4.9: Block diagram showing frequency domain equalizer two is implemented in the frequency domain in GPUs.

4.2 CPU and GPU Pipelining

The diagram and table in Figure 4.10 show how the CPU and GPUs are pipelined. The MMSE and CMA equalizer filters are computed in GPU0 (Tesla K40c GPU), because they are the most computationally heavy. Note that the MMSE equalizer filters (corresponding to the packets in a batch) are computed in the K40c, then transferred to GPU1 (Tesla K20c GPU) for filtering. This is done to maximize the GPU0 resources available for CMA iterations. The FDE1 and FDE2 equalizers are both computed and applied on GPU2 (Tesla K20c).

4.3 Laboratory Test Results

Static multipath tests were performed to assess the performance each data-aided equalizer. Figure 4.11 shows a block diagram of the configuration used for static multipath tests. The major components and their functions are summarized in Appendix A. The parameters of the multipath channel emulator were configured to produce a "three-ray" channel model motivated by the results described by Rice, Davis, and Bettwieser [22]. The model parameters are summarized in the tables in Figures 4.12 – 4.14. The receiver used for these experiments performed two functions. First, the IF output was used as the input to system described in this thesis. Second, the SOQPSK-TG demodulator was used to produce the bits for the unequalized case. Because the SOQPSK-TG demodulator was not designed to use the preamble and ASM bits, the bit decisions corresponding

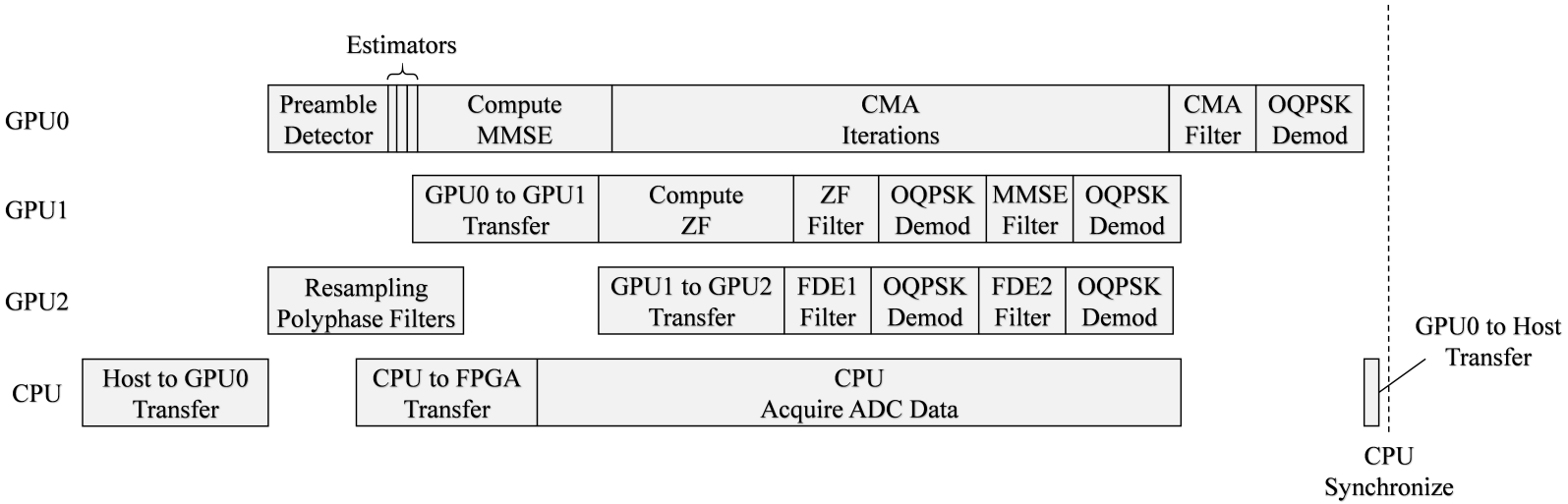


Figure 4.10: Block diagram showing how the CPU and three GPUs are pipelined.

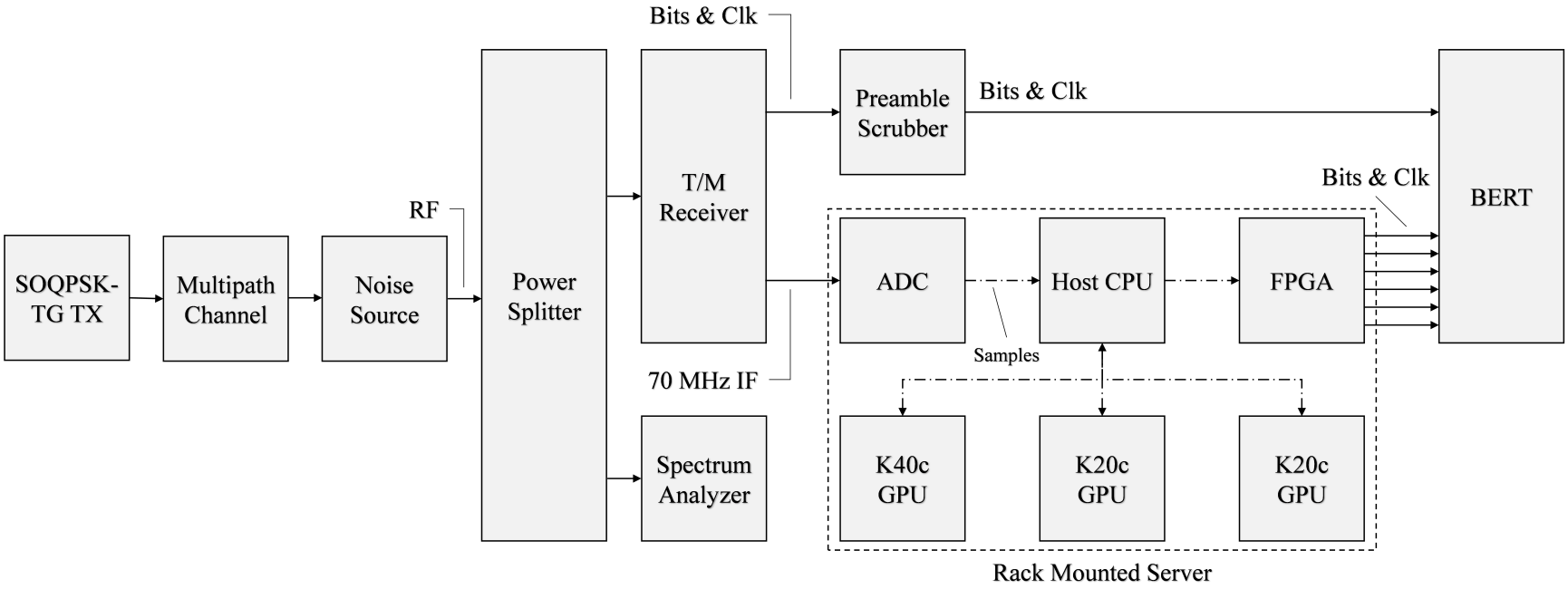


Figure 4.11: Block diagram showing the configuration for static multipath tests to compare the five data-aided equalizers to no equalization.

Table 4.5: Execution times for blocks in Figure 4.10 in order as they appear left to right then top to bottom.

Block	Execution Time (ms)
Preamble Detector	113
Estimators	18
Compute MMSE	355
CMA Iterations	1070
CMA, ZF, and MMSE Filter	44
OQPSK Demod	14
GPU0 to GPU1 Transfer	195
Compute ZF	406
Resampling Polyphase Filters	296
GPU1 to GPU2 Transfer	195
FDE1 and FDE2 Filter	58
Host to GPU0 Transfer	220
CPU to FPGA Transfer	167
CPU Acquire ADC Data	1000
GPU0 to Host Transfer	11

to the fields appeared in the demodulator output. The preamble and ASM bits in the demodulator output were removed by the preamble scrubber described by Hogstrom and Nash [23].

The BER results are summarized by the plots in Figures 4.12 – 4.14. For the channel of Figure 4.12, the ZF, MMSE, and FDE1 equalizers achieve the best performance and the CMA equalizer displays the worst performance. However all equalizers perform within about 1 dB of each other. For the channel of Figure 4.13, the CMA equalizer achieves the best performance and the FDE2 equalizer displays the worst performance. As before, all equalizers perform within about 1 dB of each other. For the channel of Figure 4.14, FDE2 achieves the best performance and CMA displays the worst performance. Here, all equalizers perform within about 2 dB of each other. Why the order of best to worst equalizer performance changes with channel parameters is not entirely clear. As a group, the performance of all of them is more similar than different. It is clear, that the BER at the equalizer output (for all the equalizers) is always better than the unequalized BER.

	Attenuation (dB)	Phase	Delay (ns)
Ray 1	0.0	0°	0
Ray 2	1.5	180°	50
Ray 3	20.0	90°	155

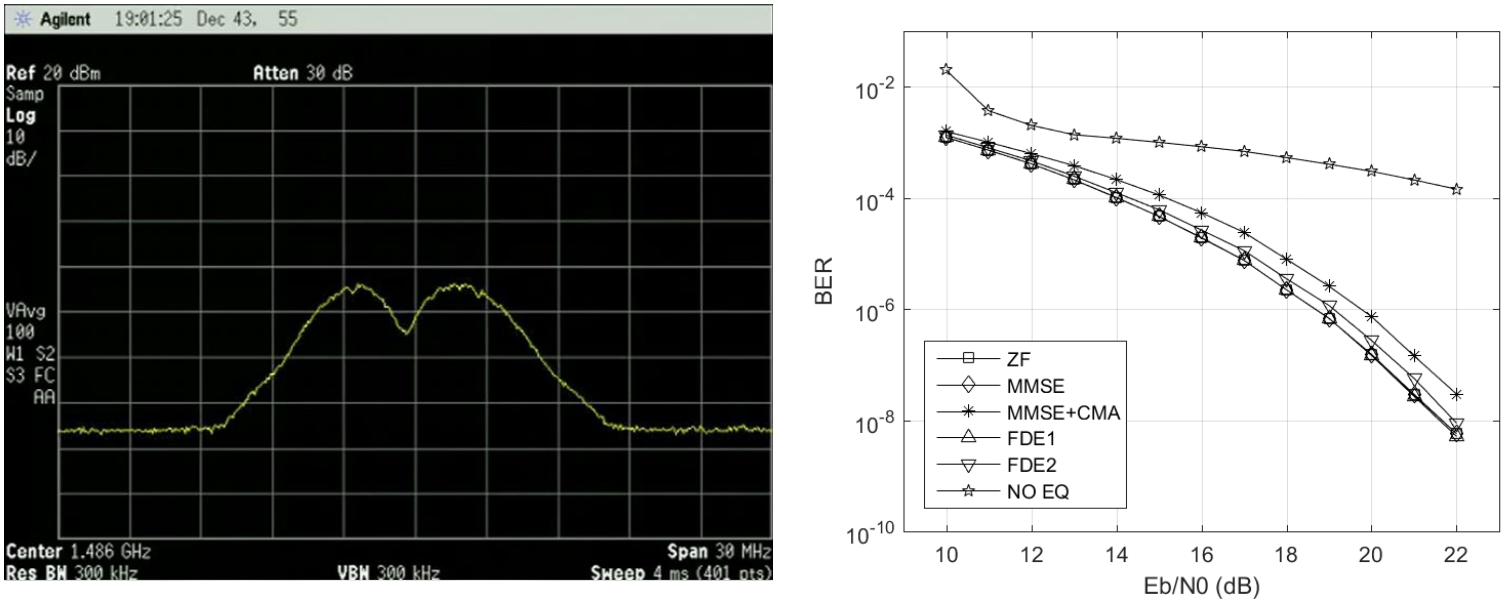


Figure 4.12: Channel 1 BER static lab test: (top) parameters for the three ray channel; (bottom left) a screen capture of the spectrum with averaging enabled; (bottom right) BER curve of the five data-aided equalizers and no equalization.

	Attenuation (dB)	Phase	Delay (ns)
Ray 1	0.0	0°	0
Ray 2	1.5	150°	50
Ray 3	20.0	90°	155

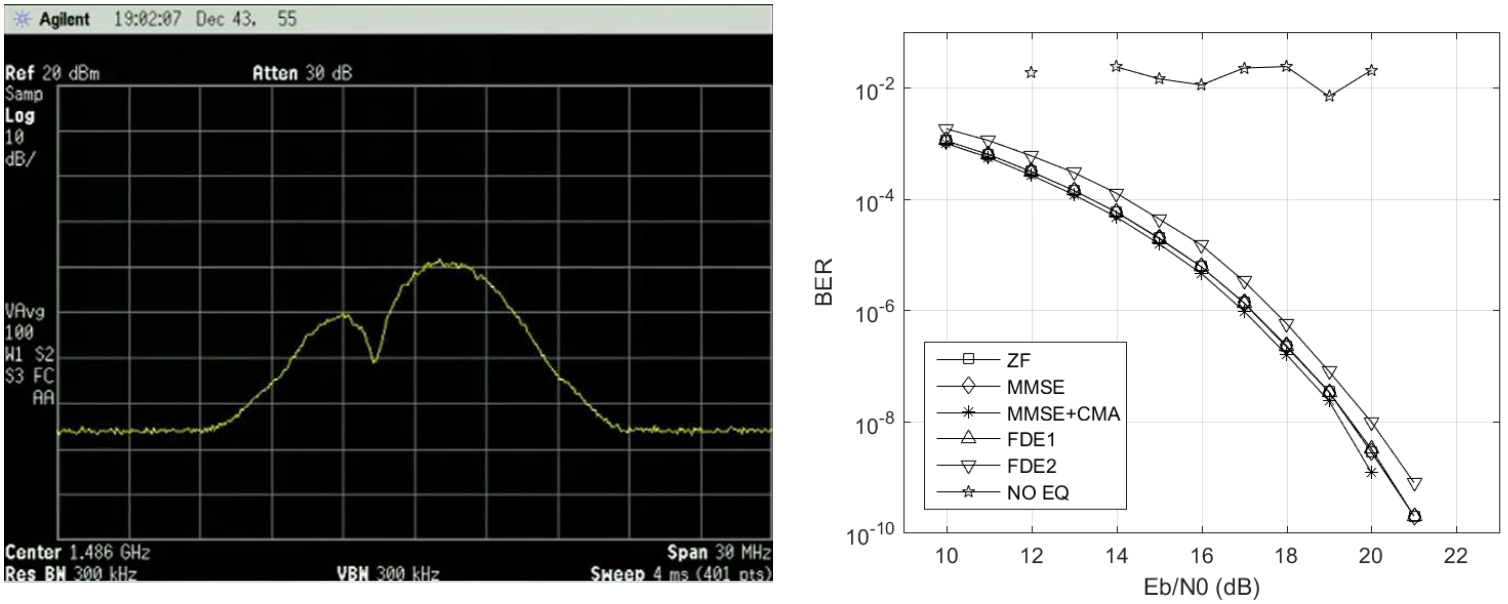
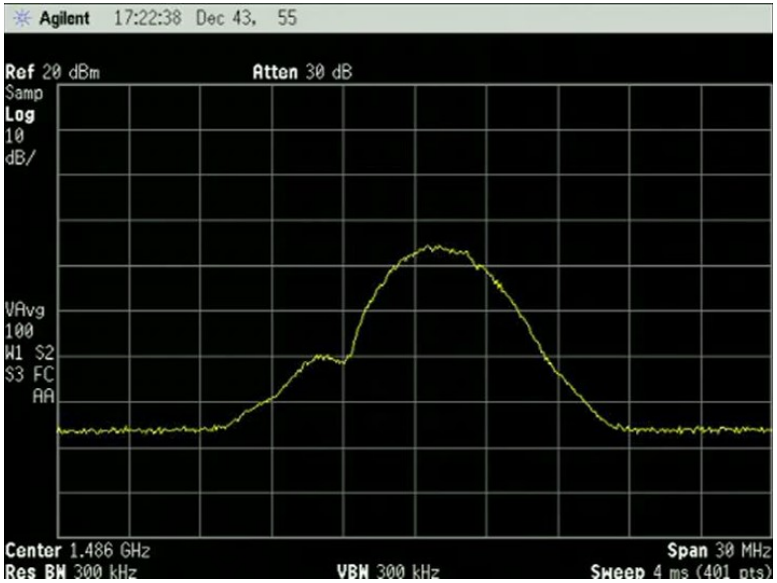


Figure 4.13: Channel 2 BER static lab test: (top) parameters for the three ray channel; (bottom left) a screen capture of the spectrum with averaging enabled; (bottom right) BER curve of the five data-aided equalizers and no equalization. No points are shown for no equalization at some values of E_b/N_0 because the demodulator was unable to lock.

	Attenuation (dB)	Phase	Delay (ns)
Ray 1	0.0	0°	0
Ray 2	1.5	120°	50
Ray 3	20.0	90°	155



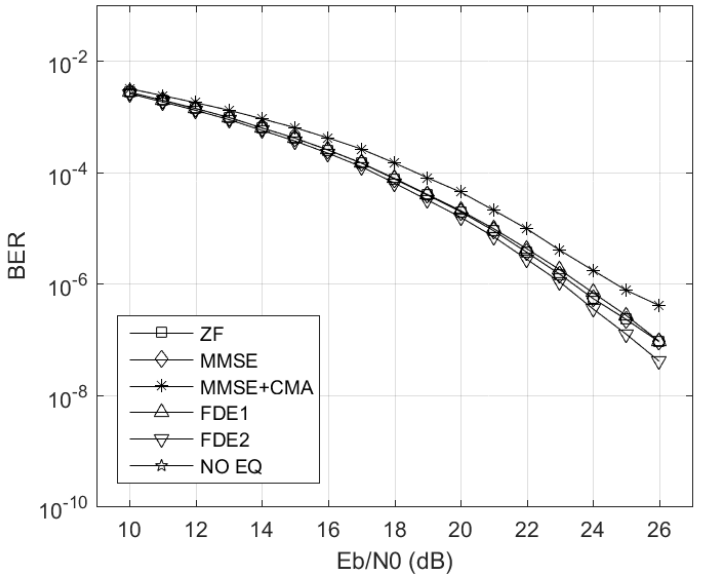


Figure 4.14: Channel 3 BER static lab test: (top) parameters for the three ray channel; (bottom left) a screen capture of the spectrum with averaging enabled; (bottom right) BER curve of the five data-aided equalizers and no equalization. No points are shown for no equalization because the demodulator was unable to lock.

CHAPTER 5. SUMMARY AND CONCLUSIONS

5.1 GPU Implementation

Based on measured execution times of GPU kernels, multiple data-aided equalization filters were implemented for the purpose of equalizing an aeronautical telemetry channel. Using GPU libraries and batch processing, rather than custom designed GPU kernels, produced massive speed ups. Also, reformulating algorithms into frequency-domain convolution produced impressive speed ups.

For implementation in one Tesla K40c and two Tesla K20c GPUs, the execution times for all equalizers met the real-time constraint. It was shown that the frequency-domain equalizers are the easiest to implement and have the fastest execution time. The CMA equalizer was shown to be the hardest to implement and has the slowest execution time. The execution time did not provide the CMA the opportunity to iterate many times. The ZF and MMSE equalizers were shown to be computationally challenging to implement, but had an acceptable execution times. Because data-aided equalizers are implemented for a real-time telemetry receiver system, the execution time results must be considered along with the bit error rate performance. The FDE1 equalizer is recommended, marking the best tradeoff between performance and computational complexity.

5.2 Contributions

The contributions of this thesis are:

1. Algorithms were implemented using batch GPU libraries.

2. GPU convolution time was reduced by using GPU libraries, batch processing, and cascading filters in the frequency domain.

3. Algorithms were implemented using linear solver libraries to make ZF and MMSE equalizers feasible and real-time.

4. The CMA equalizer was reformulated to leverage the speed of GPU convolution.

5. A new ADC to was implemented to drive success of the PAQ project.

6. Resampling polyphase filters were implemented in GPUs.

7. A paper was presented on frequency offset compensation for equalized SOQPSK at International Telemetering. Conference (ITC) [24].

5.3 Further Work

The Levinson-Durbin algorithm GPU implementation only leveraged the toeplitz structure of the channel estimate auto-correlation matrix. A hybrid sparse Levinson-Durbin algorithm could leverage the sparseness of the channel estimate auto-correlation matrix and the vector $\hat{\mathbf{g}}$.

REFERENCES

[1] M. Rice, M. S. Afran, and M. Saquib, "Equalization in aeronautical telemetry using multiple antennas," in *Proceedings of the IEEE Military Communications Conference*, Baltimore, MD, November 2014. 1

[2] M. Rice, M. S. Afran, M. Saquib, A. Cole-Rhodes, and F. Moazzami, "On the performance of equalization techniques for aeronautical telemetry," in *Proceedings of the IEEE Military Communications Conference*, Baltimore, MD, November 2014. 1, 12, 14, 16

[3] M. Rice *et al.*, "Phase 1 report: Preamble assisted equalization for aeronautical telemetry (PAQ)," Brigham Young University, Tech. Rep., 2014, submitted to the Spectrum Efficient Technologies (SET) Office of the Science & Technology, Test & Evaluation (S&T/T&E) Program, Test Resource Management Center (TRMC). Also available on-line at http://hdl.lib.byu.edu/1877/3242. 3, 14, 15

[4] integrated Network Enhanced Telemetry (iNET) Radio Access Network Standards Working Group, "Radio access network (RAN) standard, version 0.7.9," Tech. Rep., available at https://www.tena-sda.org/display/INET/iNET+Platform+Interface+Standards. 4

[5] M. Rice, *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Pearson Prentice-Hall, 2009. 4

[6] M. Rice and A. McMurdie, "On frame synchronization in aeronautical telemetry," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 5, pp. 2263–2280, October 2016. 10

[7] M. Rice and E. Perrins, "On frequency offset estimation using the inet preamble in frequency selective fading," in *Military Communications Conference (MILCOM), 2014 IEEE*. IEEE, 2014, pp. 706–711. 11

[8] H. Sari, G. Karam, and I. Jeanclaud, "Frequency-domain equalization of mobile radio and terrestrial broadcast channels," in *Global Telecommunications Conference, 1994. GLOBECOM'94. Communications: The Global Bridge., IEEE*. IEEE, 1994, pp. 1–5. 17

[9] B. Ng, C. Lam, and D. Falconer, "Turbo frequency domain equalization for single-carrier broadband wireless systems," *IEEE transactions on wireless communications*, vol. 6, no. 2, 2007. 17

[10] N. Al-Dhahir, M. Uysal, and H. Mheidat, "Single-carrier frequency domain equalization," 2008. 17

[11] J. Proakis and M. Salehi, *Digital Communications*, 5th ed. New York: McGraw-Hill, 2008. 17

[12] J. Coon, M. Sandell, M. Beach, and J. McGeehan, "Channel and noise variance estimation and tracking algorithms for unique-word based single-carrier systems," *IEEE Transactions on Wireless Communications*, vol. 5, no. 6, pp. 1488–1496, June 2006. 17

[13] I. E. Williams and M. Saquib, "Linear frequency domain equalization of SOQPSK-TG for wideband aeronautical telemetry channels," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 49, no. 1, pp. 640–647, 2013. 18

[14] E. Perrins, "FEC systems for aeronautical telemetry," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 49, no. 4, pp. 2340–2352, October 2013. 19

[15] Wikipedia, "Graphics processing unit," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Graphics_processing_unit 21

[16] ——, "Fastest fourier transform in the west," 2017. [Online]. Available: http://www.fftw.org/ 37

[17] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965. 37

[18] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Optimization for performance and energy for batched matrix computations on GPUs," in *Proceedings of the 8th Workshop on General Purpose Processing using GPUs*. ACM, 2015, pp. 59–69. 43

[19] M. Hayes, *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996. 64

[20] Wikipedia, "Sparse matrix," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Sparse_matrix 65

[21] NVIDIA, "CUDA toolkit documentation," 2017. [Online]. Available: http://docs.nvidia.com/cuda/ 65

[22] M. Rice, A. Davis, and C. Bettweiser, "A wideband channel model for aeronautical telemetry," *IEEE Transactions on Aerospace & Electronic Systems*, vol. 40, no. 1, pp. 57–69, January 2004. 72

[23] C. Hogstrom, C. Nash, and M. Rice, "Locating and removing a preamble/ASM sequence in aeronautical telemetry," in *Proceedings of the International Telemetering Conference*, Glendale, AZ, October 2016. 75, 83

[24] J. Ravert and M. Rice, "On frequency offset compensation for equalized SOQPSK," in *Proceedings of the International Telemetering Conference*, Glendale, AZ, October 2016. 80

APPENDIX A. DESCRIPTION OF BER TEST SETUP

The major components from the block diagram shown in Figure 4.11 functions are summarized as follows:

- The **SOQPSK-TG Transmitter** is a PAQ enabled L-band transmitter shown in Figure A.1.
- The **Multipath Channel** emulator generates multipath interference for a given channel at RF shown in Figure A.2.
- The **Noise Source** generates calibrated additive white Gaussian noise for a given E_b/N_0 shown in Figure A.3.
- The **Power Splitter** split the power between the Spectrum Analyzer and the T/M Receiver & Demodulator shown in Figure A.4.
- The **Spectrum Analyzer** showed the power spectrum of the multipath channel shown in Figure A.5.
- The **T/M Receiver & Demodulator** produced the no equalization bits at 10.3125 Mbits/s and down-converted RF to 70 MHz IF shown in Figure 2.4.
- The **Preamble Scrubber** explained in [23] locates and removes the preamble and ASM in the 10.3125 Mbits/s to produce a data bit stream at 10 Mbits/s shown in Figure A.6.
- The **BERT** computes the BER for six bit streams shown in Figure 2.4.

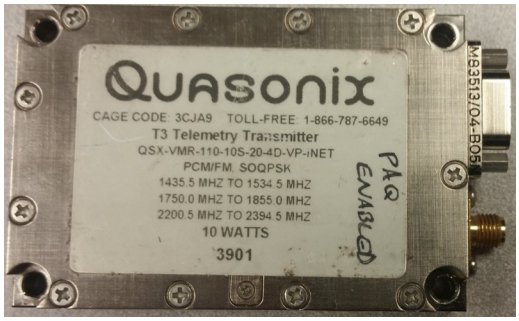


Figure A.1: PAQ enabled L-band transmitter QSX VMR 110 00S 20 2D VP iNET S/N 3901.

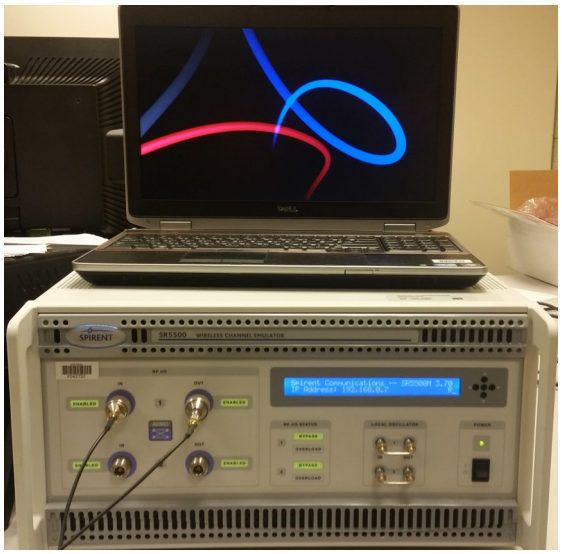


Figure A.2: Channel emulator Spirint SR 5500.

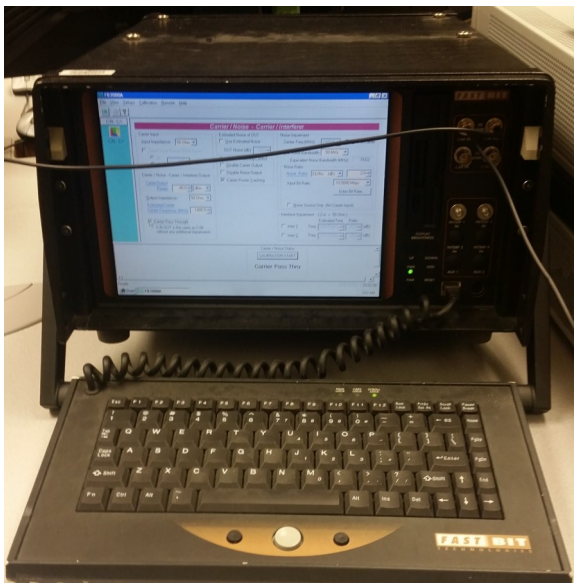


Figure A.3: Noise source Fast Bit FB0008.

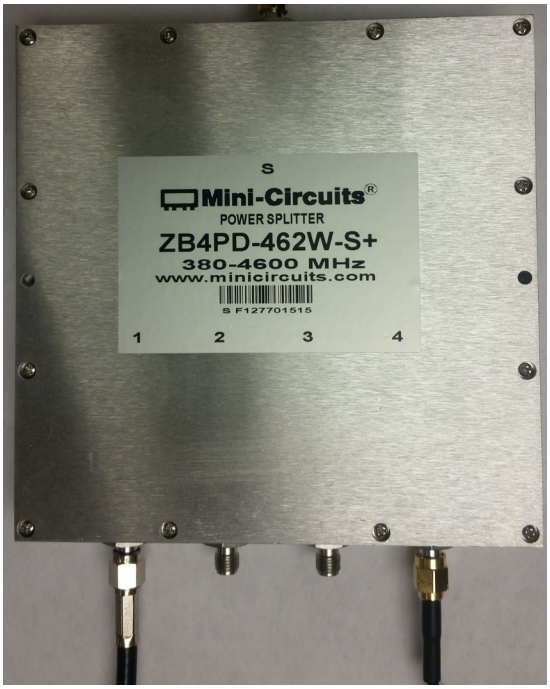


Figure A.4: Power splitter Mini-circuits ZB4PD-462W-S+.

Figure A.5: Spectrum analyzer Agilent E4404B ESA-E Series Spectrum Analyzer.

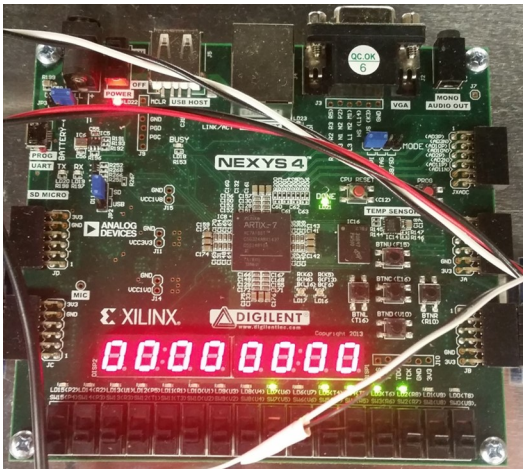


Figure A.6: Preamble and ASM scrubber.