

# GPU Implementation of Data-Aided Equalizers

Jeffrey T. 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 D. Rice, Chair  
Brian D. Jeffs  
Brian A. Mazzeo

Department of Electrical and Computer Engineering

Brigham Young University

April 2017

Copyright © 2017 Jeffrey T. Ravert

All Rights Reserved



## ABSTRACT

### GPU Implementation of Data-Aided Equalizers

Jeffrey T. Ravert

Department of Electrical and Computer Engineering

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 Constant Modulus Algorithm (CMA) equalizers are currently being used on SOQPSK-TG. The Preamble Assisted Equalization (PAQ) has been funded by the Air Force to study data-aided equalizers on SOQPSK-TG. PAQ compares side by side no equalization, data-aided zero forcing equalization, data-aided MMSE equalization, data-aided initialized CMA equalization, data-aided frequency domain equalization, and blind CMA equalization. A real time experimental test setup has been assembled including an RF receiver for data acquisition, FPGA for hardware interfacing and buffering, GPUs for signal processing, spectrum analyzer for viewing multipath events, and an 8 channel bit error rate tester to compare equalization performance. Lab tests were done with channel and noise emulators. Flight tests were conducted in March 2016 and June 2016 at Edwards Air Force Base to test the equalizers on live signals. The test setup achieved a 10Mbps throughput with a 6 second delay. This thesis studies the GPU implementation of computing and application data-aided FIR equalizer filters.

Keywords: MISSING



## **ACKNOWLEDGMENTS**

Students may use the acknowledgments page to express appreciation for the committee members, friends, or family who provided assistance in research, writing, or technical aspects of the dissertation, thesis, or selected project. Acknowledgments should be simple and in good taste.



# Table of Contents

<b>List of Tables</b>	<b>ix</b>
<b>List of Figures</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Multipath in Aeronautical Telemetry . . . . .	1
1.2 Problem Statement . . . . .	1
1.3 Organization . . . . .	2
<b>2 Preamble Assisted Equalization Project</b>	<b>3</b>
2.1 Hardware Overview . . . . .	3
2.2 Digital Signal Processing Overview . . . . .	6
2.3 Detailed Description of Signal Processing Algorithms . . . . .	9
2.3.1 Preamble Detection . . . . .	9
2.3.2 Frequency Offset Compensation . . . . .	11
2.3.3 Channel Estimation . . . . .	12
2.3.4 Noise Variance Estimation . . . . .	13
2.3.5 Symbol-by-Symbol Detector . . . . .	13
2.4 Detailed Description of Data-aided Equalizers . . . . .	15
2.4.1 Zero-Forcing and Minimum Mean Square Error Equalizers . . . . .	15
2.4.2 The Constant Modulus Algorithm . . . . .	18

2.4.3	The Frequency Domain Equalizers . . . . .	23
<b>3</b>	<b>Signal Processing in GPUs</b>	<b>27</b>
3.1	GPU and CUDA Introduction . . . . .	28
3.1.1	An Example Comparing CPU and GPU . . . . .	28
3.1.2	GPU kernel using threads and thread blocks . . . . .	31
3.1.3	GPU Memory . . . . .	32
3.1.4	Thread Optimization . . . . .	33
3.1.5	CPU and GPU Pipelining . . . . .	37
3.2	GPU Convolution . . . . .	42
3.2.1	Floating Point Operation Comparison . . . . .	43
3.2.2	CPU and GPU Single convolution using batch processing Comparison . . .	44
3.2.3	Convolution Using Batch Processing . . . . .	51
<b>4</b>	<b>Equalizer GPU Implementation</b>	<b>71</b>
4.1	Zero-Forcing and MMSE GPU Implementation . . . . .	73
4.2	Constant Modulus Algorithm GPU Implementation . . . . .	76
4.3	Frequency Domain Equalizer One and Two GPU Implementation . . . . .	77
4.3.1	Frequency Domain Equalizer One . . . . .	78
4.3.2	Frequency Domain Equalizer Two . . . . .	79
<b>5</b>	<b>Summary and Conclusions</b>	<b>81</b>
5.1	GPU Implementation . . . . .	81
5.2	Contributions . . . . .	81
5.3	Further Work . . . . .	82
	<b>Bibliography</b>	<b>83</b>

## List of Tables

2.1	CMA	22
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.	35
3.2	Defining start and stop lines for timing comparison in Listing 3.5.	47
3.3	Convolution computation times with signal length 12672 and filter length 186 on a Tesla K40c GPU.	49
3.4	Convolution computation times with signal length 12672 and filter length 21 on a Tesla K40c GPU.	50
3.5	Defining start and stop lines for execution time comparison in Listing 3.6.	52
3.6	Convolution using batch processing execution times with for a 12672 sample signal and 186 tap filter on a Tesla K40c GPU.	53
3.7	Convolution using batch processing execution times with for a 12672 sample signal and 21 tap filter on a Tesla K40c GPU.	53
3.8	Batched convolution execution times with for a 12672 sample signal and cascaded 21 and 186 tap filter on a Tesla K40c GPU.	55
4.1	Three different algorithms were explored to compute the ZF and MMSE equalizer filters.	75
4.2	Two different algorithms were explored to compute the cost function gradient $\nabla J$ .	77
4.3	Execution times for calculating and applying Frequency Domain Equalizer One and Two.	79



## List of Figures

1.1	Multipath can occur when a signal is received multiple paths like line-of-sight or ground bounce or reflections. . . . .	2
2.1	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. . . . .	4
2.2	A picture of the physical PAQ hardware referencing blocks from Figure 2.1. Right: Components in the dashed and dotted box. Left: Components in the dashed box. Note that the T/M Receiver is not pictured. . . . .	5
2.3	A diagram showing PAQ packetized sample structure. . . . .	6
2.4	Received signal has multipath interference, frequency offset, phase offset and additive white Gaussian noise. The received signal is down-converted filtered and sampled to produce the sample sequence $r(n)$ . . . . .	7
2.5	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. . . . .	7
2.6	A block diagram of the estimators in PAQ. . . . .	7
2.7	A block diagram of the computation and application of the equalizer and detection filters. The bold box emphasizes in the focus of this thesis. . . . .	8
2.8	“Numerically optimized” SOQPSK detection filter $\mathbf{h}_{\text{NO}}$ . . . . .	14
2.9	Offset Quadrature Phase Shift Keying symbol by symbol detector. . . . .	14
2.10	Diagram showing the relationships between $z(n)$ , $\rho(n)$ and $m(n)$ . . . . .	21
2.11	A diagram showing how the iNET packet is used as a cyclic prefix. . . . .	23
2.12	SOQPSK-TG power spectral density. . . . .	25
3.1	NVIDIA Tesla K40c and K20c. . . . .	28

3.2	A block diagram of how a CPU sequentially performs vector addition. . . . .	29
3.3	A block diagram of how a GPU performs vector addition in parallel. . . . .	29
3.4	32 threads launched in 4 thread blocks with 8 threads per block. . . . .	32
3.5	36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads. . . . .	32
3.6	Diagram comparing memory size and speed. Global memory is massive but extremely slow. Registers are extremely fast but there are very few. . . . .	33
3.7	Example of an NVIDIA GPU card. The GPU chip with registers, L1 cache and shared memory is shown in the dashed box. The L2 cache and global memory is shown off chip in the solid boxes. . . . .	34
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. . . . .	34
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. . . . .	37
3.10	Plot showing the number of threads per block doesn't always drastically affect execution time. . . . .	38
3.11	The typical approach of CPU and GPU operations. This block diagram shows the profile of Listing 3.3. . . . .	38
3.12	GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 3.4. . . . .	39
3.13	A block diagram of pipelining a CPU with three GPUs. . . . .	40
3.14	Block diagrams showing time-domain convolution and frequency-domain convolution. . . . .	43
3.15	Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 186 tap complex filter. . . . .	45
3.16	Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 21 tap complex filter. . . . .	46
3.17	Comparison of number of floating point operations (flops) required to convolve a 12672 sample complex signal with a variable length tap complex filter. . . . .	47

3.18 Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 186 taps. The comparison is messy with out lower bounding. . . . .	48
3.19 Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 186 taps. A lower bound was applied by searching for a local minimums in 15 sample width windows. . . . .	49
3.20 Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 21 taps. A lower bound was applied by searching for a local minimums in 5 sample width windows. . . . .	50
3.21 Comparison of a complex convolution on CPU verse GPU. The filter length is variable and the signal is fixed at 12672 samples. A lower bound was applied by searching for a local minimums in 3 sample width windows. . . . .	51
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 12672 and 186. . . .	53
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 12672 and 186. . . .	54
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. . . . .	55
3.25 Comparison of complex convolution using batch processing on a GPU. The signal length is variable and the filter is fixed at 21 taps. . . . .	56
3.26 Comparison of complex convolution using batch processing on a GPU. The filter length is variable and the signal length is set to 12672 samples. . . . .	57
3.27 Block diagrams showing cascaded time-domain convolution and frequency-domain convolution. . . . .	58
4.1 To simplify block diagrams, frequency-domain convolution is shown as one block. . . . .	72
4.2 To simplify block diagrams, frequency-domain cascaded convolution is shown as one block. . . . .	72
4.3 Block Diagram showing how the Zero-Forcing equalizer coefficients are implemented in the GPU. . . . .	75
4.4 Block Diagram showing how the Minimum Mean Squared Error equalizer coefficients are implemented in the GPU. . . . .	75

4.5	Block Diagram showing how the CMA equalizer filter is implemented in the GPU using frequency-domain convolution twice per iteration. . . . .	78
4.6	Block diagram showing Frequency Domain Equalizer One is implemented in the frequency domain in GPUs. . . . .	80
4.7	Block diagram showing Frequency Domain Equalizer Two is implemented in the frequency domain in GPUs. . . . .	80

# **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 in aeronautical telemetry when the transmitted signal is received from multiple paths because 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 impulse response. Equalizers have been studied to combat multipath interference in aeronautical telemetry [1, 2].

There are two types of equalizers, blind and data-aided. Blind equalizers combat multipath using known properties of the transmitted signal but no knowledge of the data or multipath channel. Data-aided equalizers require knowing something about the received signal. One method of providing data that can be used in data-aided equalization is for the transmitter to periodically insert a known bit sequence called a “pilot” into the data stream. The receiver compares the received signal with a locally stored copy to estimate parameters such as multipath channels, frequency offsets, phase offsets and noise variance. Data-aided equalizers are finite-length impulse response (FIR) filters. The impulse response of the equalizer filter is computed based on the estimated channel better mitigate multipath.

### **1.2 Problem Statement**

The signal processing for a digital communications system with data-aided equalizers is computationally heavy. Algorithms implemented in a high powered Central Processing Unit (CPU) can not process in real-time. Graphic Processing Units (GPUs) can be used to perform real-time processing because of their massively parallel architecture.

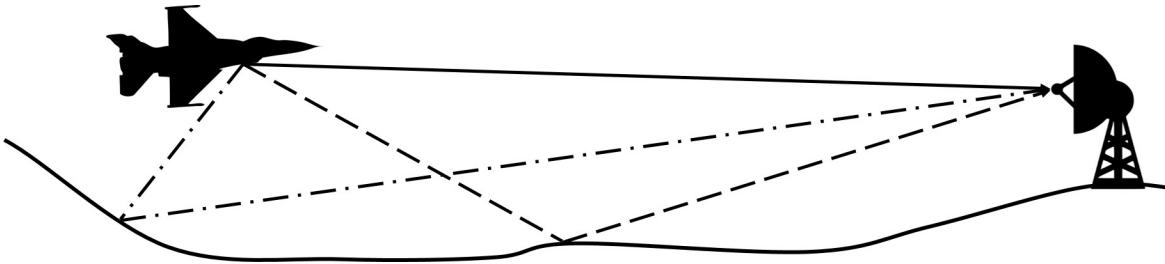


Figure 1.1: Multipath can occur when a signal is received multiple paths like line-of-sight or ground bounce or reflections.

This thesis studies how signal processing can be reformulated to run quickly and efficiently in GPUs. Processing signals in batches to introduce more parallelism. Optimized libraries harness GPU resources to make signal processing implementation relatively easy and extremely fast. If algorithms can be reformulated to used matrix/vector multiplication, solving linear systems of equations, or the Fast Fourier Transform, GPUs can provide vast speed ups.

### 1.3 Organization

Chapter ?? shows the equations for these block diagrams. Chapter 3 will shed some light on signal processing in GPUs. Chapter 4 will illustrate how the five equalizers are implemented in GPUs. Chapter 5 will summarize.

## Chapter 2

### Preamble Assisted Equalization Project

Data-aided equalization in aeronautical telemetry has been studied and tested by the Preamble Assisted Equalization (PAQ) project [3]. PAQ built a system that compares five data-aided equalizers to blind equalization and no equalization. Live flight tests were conducted at Edwards AFB in March and June 2016. The five data-aided equalizers PAQ studied are

- zero-forcing (ZF) equalizer
- minimum mean square Error (MMSE) equalizer
- MMSE initialized constant modulus algorithm (CMA) equalizer
- frequency domain equalizer one (FDE1)
- frequency domain equalizer Two (FDE2).

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

#### 2.1 Hardware Overview

A block diagram of PAQ physical system is shown in Figure 2.1. A picture of the physical components is shown in Figure 2.2. The major components, and their functions are summarized in the following.

- The **T/M receiver** down-converts from L- or C-band RF to 70 MHz IF then applies an anti-aliasing filter.
- 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.

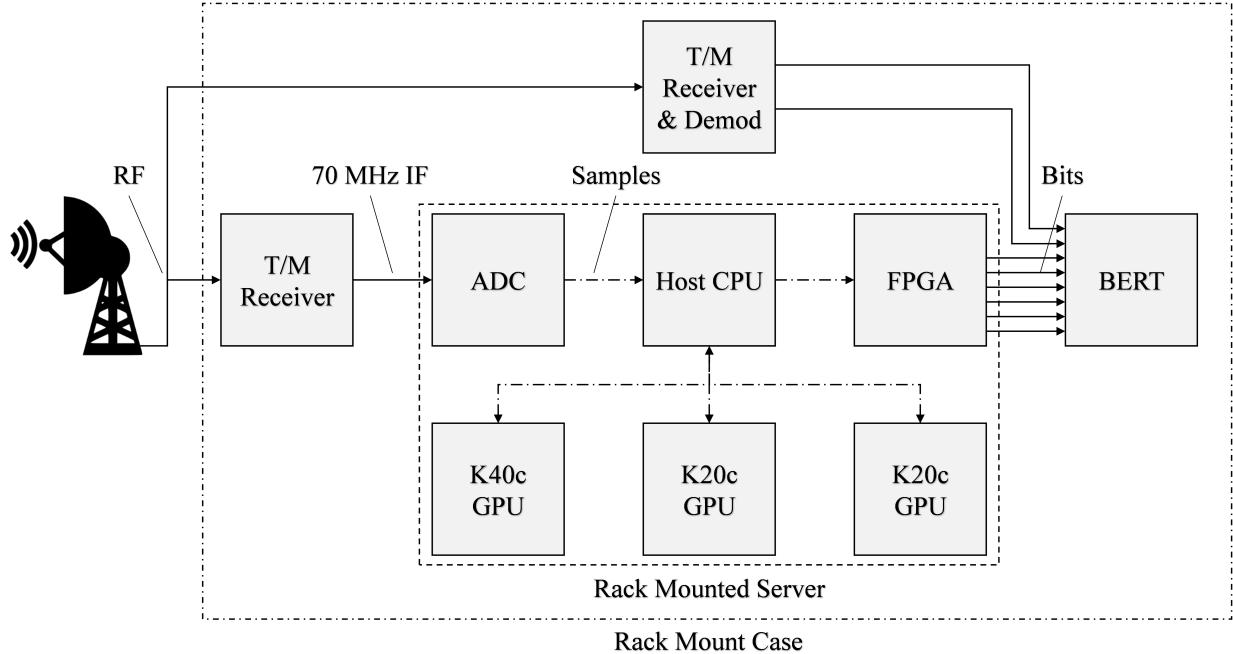


Figure 2.1: A block diagram of the physical PAQ hardware. The components inside the rack mounted server are in the dashed box. All the components in the dashed and dotted box are housed in a rack mounted case.

- The **ADC** produces 14-bit samples of the real-valued bandpass signal centered at IF. 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.
- The bit error rate tester (**BERT**) counts the errors in each input bit stream by comparing the streams to a 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.

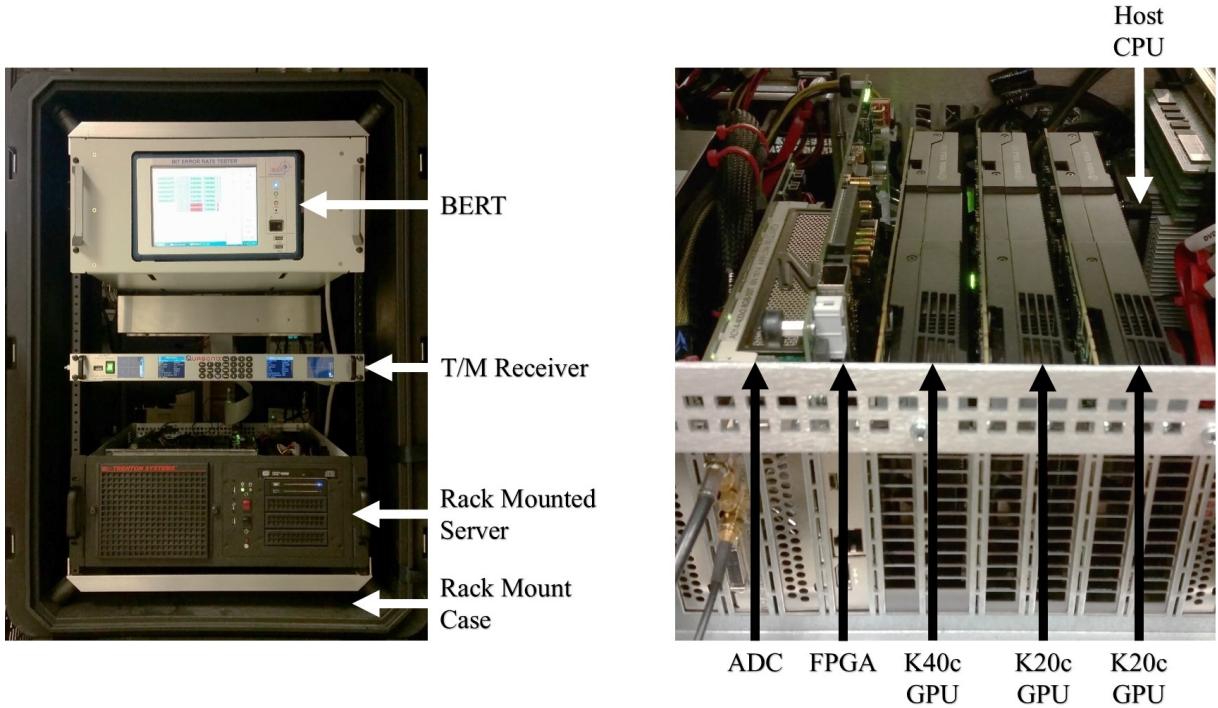


Figure 2.2: A picture of the physical PAQ hardware refrencing blocks from Figure 2.1. Right: Components in the dashed and dotted box. Left: Components in the dashed box. Note that the T/M Receiver is not pictured.

- The **T/M Receiver & Demod** demodulates the RF signal outputting two bit streams for blind equalization and no equalization for BER comparison.

To enable data-aided equalization, PAQ bit stream has a packetized structure shown in Figure 2.3. The bit stream has a pilot bit sequence, in the form of the iNET preamble and ASM, periodically inserted into the data bits. The iNET preamble comprises eight repetitions of the 16-bit sequence  $CD98_{hex}$  and the ASM field is

$$034776C7272895B0_{hex}. \quad (2.1)$$

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 bits are modulated by a SOQPSK-TG modulation scheme at 10 Mbits/second. With the preamble and ASM periodically inserted, the over-the-air bit rate is 10.3125 Mbits/second.

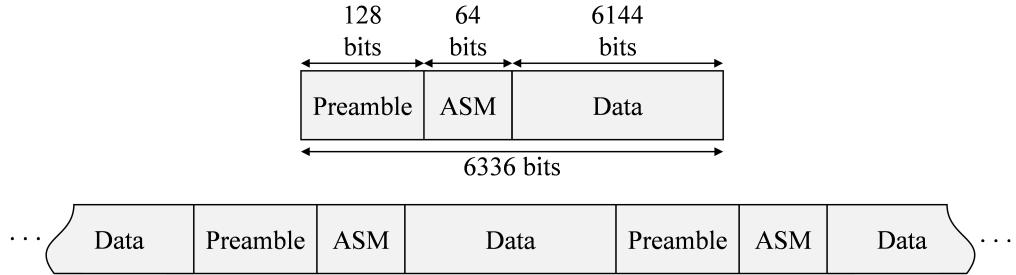


Figure 2.3: A diagram showing PAQ packetized sample structure.

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  $\omega_0$ , a phase offset  $\phi$  and additive white Gaussian noise  $w(t)$ . The received signal is down-converted, filtered in the T/M receiver, sampled at  $93^{1/3}$  Msamples/second by the ADC then down-converted to baseband and resampled by  $^{99}/448$  in the GPUs in one step using a polyphase filter bank based on the principles outlined in [4, 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.

The model of the received signal is shown in Figure 2.4. At baseband and 2 samples/bit, the FIR channel impulse response is assumed to have a non-causal component comprising  $N_1$  samples and a causal component comprising  $N_2$  samples. Figure 2.5 shows the full discrete-time  $L_h = N_1 + N_2 + 1$  sample channel. The iNET packet is  $L_{\text{pkt}} = 12672$  samples long with the preamble  $L_p = 256$  samples and the ASM  $L_{\text{ASM}} = 128$  samples.

## 2.2 Digital Signal Processing Overview

A high-level digital signal processing flow is shown in Figure 2.6 and 2.7. Because the frequency offset, channel, and noise variance are estimated using the preamble and ASM, the first step is to find the samples correlating to the preamble in the received sample sequence  $r(n)$ . The preamble detector block outputs the vector of samples  $\mathbf{r}_p$  with the iNET packetized structure. The first  $L_p + L_{\text{ASM}}$  samples in  $\mathbf{r}_p$  correlate with the received pilot samples.

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  $\mathbf{r}_p$  to produce  $\mathbf{r}$ . The

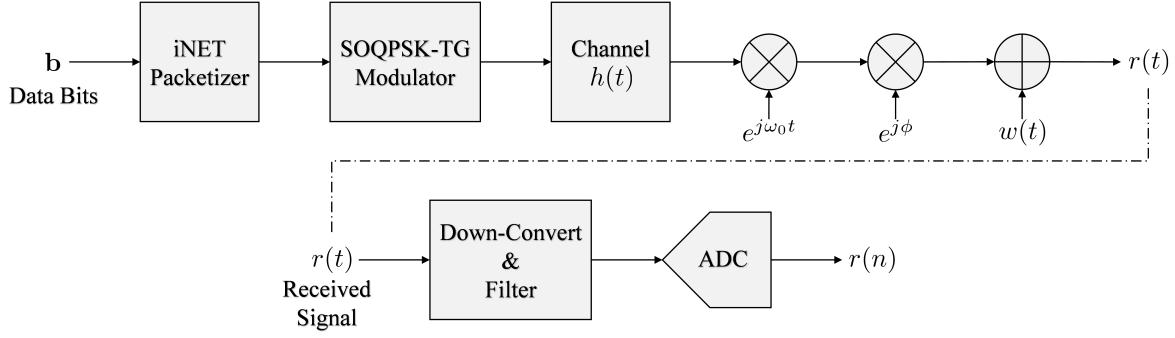


Figure 2.4: Received signal has multipath interference, frequency offset, phase offset and additive white Gaussian noise. The received signal is down-converted filtered and sampled to produce the sample sequence  $r(n)$ .

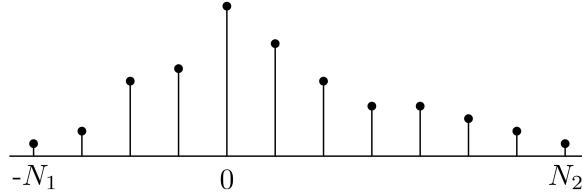


Figure 2.5: An illustration of the discrete-time channel of length  $N_1 + N_2 + 1$  with a non-causal component comprising  $N_1$  samples and a causal component comprising  $N_2$  samples.

de-rotated preamble and ASM samples in the vector  $\mathbf{r}$  are used to estimate the channel  $\hat{\mathbf{h}}$  and noise variance  $\hat{\sigma}_w^2$ . The channel and noise variance estimates are computed in the estimators block.

Equipped with knowledge of the estimated channel and noise variance estimates, the impulse response of the equalizer filters are computed. Figure 2.7 shows a block diagram of the five independent branches, each branch computing an equalizer filter and producing a vector of estimated bits  $\hat{\mathbf{b}}$ . The figure shows each equalizer requires different estimated vectors.

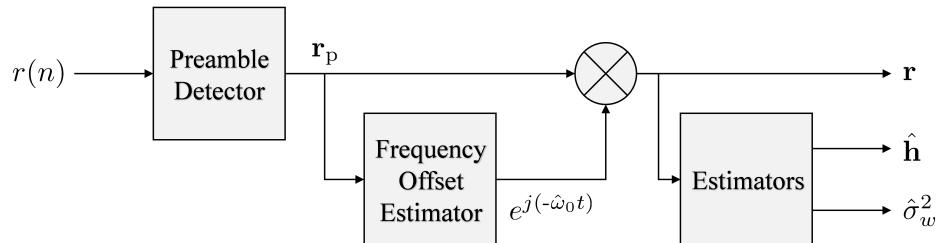


Figure 2.6: A block diagram of the estimators in PAQ.

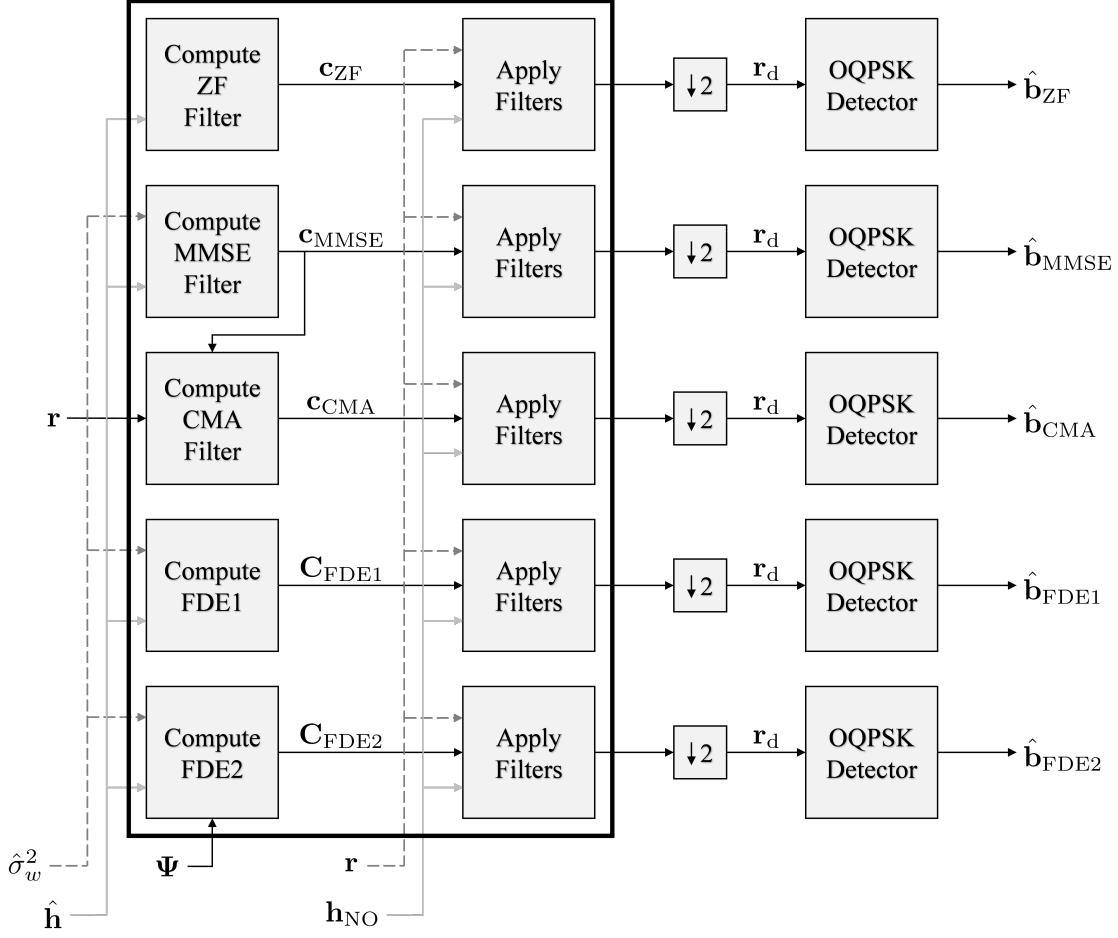


Figure 2.7: A block diagram of the computation and application of the equalizer and detection filters. The bold box emphasizes in the focus of this thesis.

- The ZF equalizer filter  $c_{ZF}$  only requires the channel estimate  $\hat{h}$ .
- The MMSE equalizer filter  $c_{MMSE}$  requires the channel estimate  $\hat{h}$  and the noise variance estimate  $\hat{\sigma}_w^2$ .
- The CMA equalizer filter  $c_{CMA}$  requires the de-rotated received signal  $r$  and the MMSE equalizer filter  $c_{MMSE}$  for initialization.
- The FDE1 equalizer in the frequency domain  $C_{FDE1}$  requires the channel estimate  $\hat{h}$  and the noise variance estimate  $\hat{\sigma}_w^2$ .

- The FDE2 equalizer in the frequency domain  $C_{FDE2}$  requires the power spectral density of SOQPSK-TG  $\Psi$ , the channel estimate  $\hat{h}$  and the noise variance estimate  $\hat{\sigma}_w^2$ .

Each branch filters the de-rotated received signal  $r$  with the computed equalizer filter and the detection filter  $h_{NO}$ .

The equalizer filters are FIR filters whose impulse response has support  $-L_1 \leq n \leq L_2$  (i.e. the equalizer length is  $L_{EQ} = L_1 + L_2 + 1$  samples). The length of the equalizers was batched to five times the length of the channel. In particular the non-causal component comprising  $L_1 = 5N_1 = 60$  samples and a causal component comprising  $L_2 = 5N_2 = 125$  samples resulting in  $L_{EQ} = L_1 + L_2 + 1 = 186$ .

Symbol-by-symbol OQPSK detection [5, app. (A)] is performed on the down-sampled equalizer output. The  $r_d$  in each branch has a sample rate of 1 sample/bit. The OQPSK detector block outputs the vector of estimated bits  $\hat{b}$ . Finally the BER for each equalizer is obtained by comparing the vectors of estimated bits  $\hat{b}$  to the PN sequence.

The GPUs in Figure 2.1 and 2.2 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 is 1907 milliseconds worth of data. Each batch has at most 3104 12672-sample iNET packets. The GPU processes at the most 3104 packets in parallel by leveraging batched processing. Each batch at the most 3104 packets, each packet performs the algorithms shown in Figures 2.6 and 2.7. To meet the real time requirement, **all** processing must be completed in less than 1907 ms.

This thesis, illustrates how the five PAQ data-aided equalizers were computed and applied in GPUs. The bold box in Figure 2.7 emphasizes processing blocks on which this thesis focuses.

## 2.3 Detailed Description of Signal Processing Algorithms

### 2.3.1 Preamble Detection

To compute data-aided equalizers, samples of the received waveform corresponding to the preamble and ASM bits are found then used to estimate parameters. The goal of the preamble detection step is to structure the received samples  $r(n)$  into  $L_{pkt}$  sample packets denoted  $r_p$ . Each vector of samples  $r_p$  has the structure shown in Figure 2.3.

Before the structuring the received samples into packets, the preambles are found using the preamble detector described in [6]. The preamble detector computes the function  $L(n)$  for each sample in the batch. Peaks in  $L(n)$  identify the locations of the start of the preamble. The function  $L(n)$  is given by

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

where

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

and

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

with

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

A correlation peak in  $L(n)$  indicates the index  $n$  is the start of a preamble. The vector of packet samples starting at index  $n$  are

$$\mathbf{r}_p = \begin{bmatrix} r(n) \\ \vdots \\ r(n + L_{\text{pkt}} - 1) \end{bmatrix} = \begin{bmatrix} r_p(0) \\ \vdots \\ r_p(L_{\text{pkt}} - 1) \end{bmatrix} \tag{2.6}$$

### 2.3.2 Frequency Offset Compensation

The frequency offset estimator shown in Figure 2.6 is the estimator taken from [7, eq. (24)]. With the notation adjusted slightly, the frequency offset estimate is

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

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

$$r(n) = r_p(n)e^{-j\hat{\omega}_0 n}. \tag{2.8}$$

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

### 2.3.3 Channel Estimation

The channel estimate is the discrete time estimation of the multipath channel defined by the coefficients

$$\hat{h}(-N_1) \quad \dots \quad \hat{h}(0) \quad \dots \quad \hat{h}(N_2). \quad (2.9)$$

The vector of channel estimate coefficients is denoted by

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

The channel estimator is the ML estimator taken from [2, eq. 8].

$$\hat{\mathbf{h}} = \underbrace{(\mathbf{X}^\dagger \mathbf{X})^{-1} \mathbf{X}^\dagger \mathbf{r}}_{\mathbf{X}_{\text{lpi}}} \quad (2.11)$$

where

$$\mathbf{X} = \begin{bmatrix} p(N_2) & & & & & \\ \vdots & p(N_2) & & & & \\ p(L_p + L_{\text{ASM}} - N_1) & & \ddots & & & \\ & \vdots & & p(N_2) & & \\ & p(L_p + L_{\text{ASM}} - N_1) & & & \ddots & \\ & & & & & \vdots \\ & & & & & p(L_p + L_{\text{ASM}} - N_1) \end{bmatrix} \quad (2.12)$$

and is a  $(L_p + L_{\text{ASM}} - N_1 - N_2) \times (N_1 + N_2 + 1)$  convolution matrix formed from the samples of SOQPSK-TG corresponding to the preamble and ASM bits

$$\mathbf{p} = [p(0) \quad p(1) \quad \dots \quad p(L_p + L_{\text{ASM}} - 1)]. \quad (2.13)$$

The  $(N_1 + N_2 + 1) \times (L_p + L_{\text{ASM}} - N_1 - N_2)$  matrix  $\mathbf{X}_{\text{lpi}}$  is the left pseudo-inverse of  $\mathbf{X}$ . The matrix vector multiplication  $\mathbf{X}_{\text{lpi}}\mathbf{r}$  is implemented simply and efficiently in GPUs.

### 2.3.4 Noise Variance Estimation

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

$$\hat{\sigma}_w^2 = \frac{1}{2\rho} \left| \mathbf{r} - \mathbf{X}\hat{\mathbf{h}} \right|^2 \quad (2.14)$$

where

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

Equation (2.14) is easily implemented into GPUs.

### 2.3.5 Symbol-by-Symbol Detector

Symbol-by-symbol detection comprises a detection filter and a phase lock loop (PLL) to track out the residual frequency offset. Before the symbols are detected, the equalized samples are passed through the detection filter then down-sampled by 2. The detection filter is a  $L_{\text{df}} = 21$  sample “numerically optimized” SOQPSK detection filter  $\mathbf{h}_{\text{NO}}$  shown in Figure 2.8 [8, Fig. 3]. The symbol-by-symbol detector block in Figure 2.7 is a Offset Quaternary Phase Shift Keying (OQPSK) detector. Using the simple OQPSK detector in place of a complex MLSE SOQPSK-TG detector leads to less than 1 dB loss in detection efficiency [8].

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.

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 each iteration requires only 10 floating point operations and a few logic decisions.

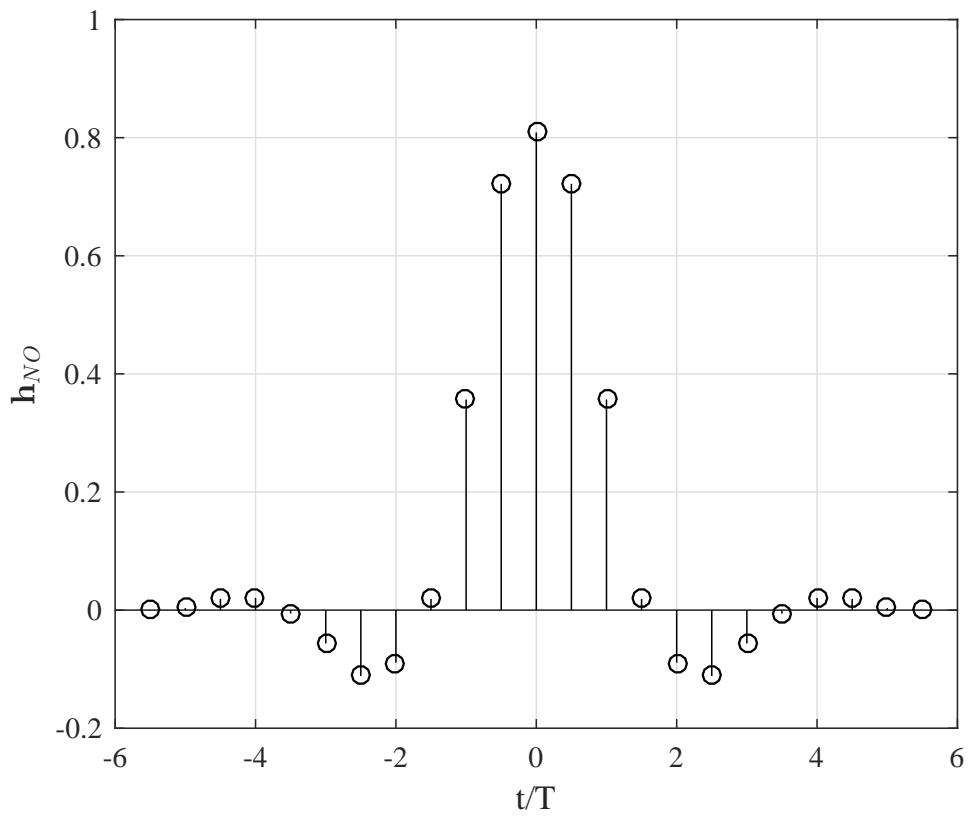


Figure 2.8: “Numerically optimized” SOQPSK detection filter  $h_{NO}$ .

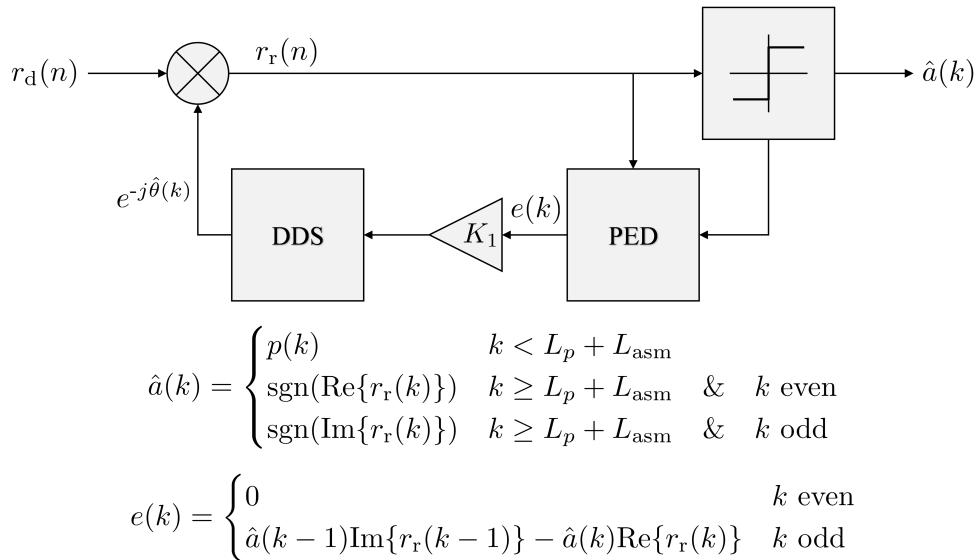


Figure 2.9: Offset Quadrature Phase Shift Keying symbol by symbol detector.

## 2.4 Detailed Description of Data-aided Equalizers

This thesis examines the GPU implementation of the five equalizer filters. While the performance and GPU implementation is interesting, this thesis makes no claim of theoretically expanding understanding of equalizers. The data-aided equalizers studied in this thesis are:

- zero-forcing (ZF) equalizer
- minimum mean square Error (MMSE) equalizer
- MMSE-initialized constant modulus algorithm (CMA) equalizer
- frequency domain equalizer one (FDE1)
- frequency domain equalizer Two (FDE2).

The final equations for ZF and MMSE FIR equalizer filters are very similar but differ in formulation. The equations for FDE1 and FDE2 are also very similar but differ by one subtle difference. The CMA FIR equalizer filter is computed using a steepest decent algorithm initialized by MMSE. More CMA iterations lead to lower BER but ZF, MMSE, FDE1 and FDE2 only require one iteration. The equations explained in this section will be referenced heavily in Chapter 4.

### 2.4.1 Zero-Forcing and Minimum Mean Square Error Equalizers

The ZF and MMSE equalizers are treated together here because they have many common features. Both equalizers are found by solving linear equations

$$\mathbf{Ac} = \mathbf{b} \quad (2.16)$$

where  $\mathbf{c}$  is a vector of desired equalizer coefficients and the square matrix  $\mathbf{A}$  and vector  $\mathbf{b}$  are known. It will be shown that the only difference between ZF and MMSE lies in the matrix  $\mathbf{A}$ .

#### Zero-Forcing

The ZF equalizer is an FIR filter defined by the coefficients

$$c_{\text{ZF}}(-L_1) \quad \dots \quad c_{\text{ZF}}(0) \quad \dots \quad c_{\text{ZF}}(L_2). \quad (2.17)$$

The filter coefficients are the solution to the matrix vector equation [5, eq. (311)]

$$\mathbf{R}_{\hat{h}} \mathbf{c}_{\text{ZF}} = \hat{\mathbf{h}}_{n_0} \quad (2.18)$$

where

$$\mathbf{c}_{\text{ZF}} = \begin{bmatrix} c_{\text{ZF}}(-L_1) \\ \vdots \\ c_{\text{ZF}}(0) \\ \vdots \\ c_{\text{ZF}}(L_2) \end{bmatrix}, \quad (2.19)$$

$$\mathbf{R}_{\hat{h}} = \mathbf{H}^\dagger \mathbf{H} = \begin{bmatrix} r_{\hat{h}}(0) & r_{\hat{h}}^*(1) & \cdots & r_{\hat{h}}^*(L_{eq}-1) \\ r_{\hat{h}}(1) & r_{\hat{h}}(0) & \cdots & r_{\hat{h}}^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_{\hat{h}}(L_{eq}-1) & r_{\hat{h}}(L_{eq}-2) & \cdots & r_{\hat{h}}(0) \end{bmatrix} \quad (2.20)$$

and

$$\hat{\mathbf{h}}_{n_0} = \mathbf{H}^\dagger \mathbf{u}_{n_0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix}, \quad (2.21)$$

where

$$r_{\hat{h}}(k) = \sum_{n=-N_1}^{N_2} \hat{h}(n) \hat{h}^*(n-k), \quad (2.22)$$

$$\mathbf{u}_{n_0} = \left\{ \begin{array}{l} \left[ \begin{array}{c} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{array} \right] \\ \left\{ \begin{array}{l} n_0 - 1 \text{ zeros} \\ N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \end{array} \right. \end{array} \right. , \quad (2.23)$$

$n_0 = N_1 + L_1 + 1$  and

$$\mathbf{H} = \begin{bmatrix} \hat{h}(-N_1) & & & \\ \hat{h}(-N_1 + 1) & \hat{h}(-N_1) & & \\ \vdots & \vdots & \ddots & \\ \hat{h}(N_2) & \hat{h}(N_2 - 1) & \hat{h}(-N_1) & \\ & \hat{h}(N_2) & \hat{h}(-N_1 + 1) & \\ & & \vdots & \\ & & & \hat{h}(N_2) \end{bmatrix}. \quad (2.24)$$

The  $(L_{\text{EQ}} \times L_{\text{EQ}})$  auto-correlation matrix  $\mathbf{R}_{\hat{h}}$  comprises the sample auto-correlation  $r_{\hat{h}}(k)$ . Building  $\mathbf{R}_{\hat{h}}$  using  $r_{\hat{h}}(k)$  eliminates the need for the matrix matrix multiplication  $\mathbf{H}^\dagger \mathbf{H}$ . Note that the sample auto-correlation  $r_{\hat{h}}(k)$  only has support on  $-(L_{ch} - 1) \leq k \leq L_{ch} - 1$ , making the auto-correlation matrix  $\mathbf{R}_{\hat{h}}$  sparse or 63% zero.

## MMSE Equalizer

The MMSE equalizer is an FIR filter defined by the coefficients

$$c_{\text{MMSE}}(-L_1) \quad \cdots \quad c_{\text{MMSE}}(0) \quad \cdots \quad c_{\text{MMSE}}(L_2). \quad (2.25)$$

The filter coefficients are the solution to the matrix vector equation [5, eq. (330) and (333)]

$$\mathbf{R}\mathbf{c}_{\text{MMSE}} = \mathbf{g}^\dagger \quad (2.26)$$

where

$$\mathbf{c}_{\text{MMSE}} = \begin{bmatrix} c_{\text{MMSE}}(-L_1) \\ \vdots \\ c_{\text{MMSE}}(0) \\ \vdots \\ c_{\text{MMSE}}(L_2) \end{bmatrix}, \quad (2.27)$$

$$\mathbf{R} = \mathbf{G}\mathbf{G}^\dagger + 2\hat{\sigma}_w^2 \mathbf{I}_{L_1+L_2+1} = \begin{bmatrix} r_h(0) + 2\hat{\sigma}_w^2 & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) + 2\hat{\sigma}_w^2 & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \vdots \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) + 2\hat{\sigma}_w^2 \end{bmatrix} \quad (2.28)$$

and

$$\mathbf{g}^\dagger = \hat{\mathbf{h}}_{n0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix}, \quad (2.29)$$

where

$$r_{\hat{h}}(k) = \sum_{n=-N_1}^{N_2} \hat{h}(n)\hat{h}^*(n-k) \quad (2.30)$$

and

$$\mathbf{G} = \begin{bmatrix} \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \\ \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \\ \ddots & & \ddots \\ & & \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \end{bmatrix}. \quad (2.31)$$

The  $(L_{EQ} \times L_{EQ})$  auto-correlation matrix  $\mathbf{R}$  comprises the sample auto-correlation  $r_{\hat{h}}(k)$  and the noise variance estimate  $\hat{\sigma}_w^2$ . Building  $\mathbf{R}$  using  $r_{\hat{h}}(k)$  and  $\hat{\sigma}_w^2$  eliminates the need for the matrix matrix multiplication  $\mathbf{G}\mathbf{G}^\dagger$ . Note that the sample auto-correlation  $r_{\hat{h}}(k)$  only has support on  $-(L_{ch}-1) \leq k \leq L_{ch}-1$  and the addition of  $\hat{\sigma}_w^2$  does not increase that support, making the auto-correlation matrix  $\mathbf{R}$  sparse or 63% zero.

## 2.4.2 The Constant Modulus Algorithm

The  $b$ th CMA equalizer is an FIR filter defined by the coefficients

$$c_{\text{CMA}}^b(-L_1) \quad \cdots \quad c_{\text{CMA}}^b(0) \quad \cdots \quad c_{\text{CMA}}^b(L_2). \quad (2.32)$$

The filter coefficients are calculated by a steepest decent algorithm

$$\mathbf{c}_{\text{CMA}}^{b+1} = \mathbf{c}_{\text{CMA}}^b - \mu \nabla J \quad (2.33)$$

initialized by the MMSE equalizer coefficients

$$\mathbf{c}_{\text{CMA}}^0 = \mathbf{c}_{\text{MMSE}}. \quad (2.34)$$

The vector  $\mathbf{J}$  is the cost function and  $\nabla J$  is the cost function gradient [5, eq. (352)]

$$\nabla J = \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \left[ y(n)y^*(n) - 1 \right] y(n) \mathbf{r}^*(n). \quad (2.35)$$

where

$$\mathbf{r}(n) = \begin{bmatrix} r(n + L_1) \\ \vdots \\ r(n) \\ \vdots \\ r(n - L_2) \end{bmatrix}. \quad (2.36)$$

This means  $\nabla J$  is defined by

$$\nabla J = \begin{bmatrix} \nabla J(-L_1) \\ \vdots \\ \nabla J(0) \\ \vdots \\ \nabla J(L_2) \end{bmatrix}. \quad (2.37)$$

A DSP engineer could implement the steepest decent algorithm by computing the cost function gradient directly. The  $L_{pkt}$  sample summation for  $\nabla J$  in (2.35) does not map well to GPUs. Chapter 3 will show how well convolution performs in GPUs. The computation for  $\nabla J$  can be massaged and re-expressed as convolution.

To begin messaging  $\nabla J$ , the term

$$z(n) = 2 \left[ y(n)y^*(n) - 1 \right] y(n) \quad (2.38)$$

is defined to simplify the expression of  $\nabla J$  to

$$\nabla J = \frac{1}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} z(n) \mathbf{r}^*(n). \quad (2.39)$$

Expanding the expression of  $\nabla J$  into vector form

$$\nabla J = \frac{z(0)}{L_{pkt}} \begin{bmatrix} r^*(L_1) \\ \vdots \\ r^*(0) \\ \vdots \\ r^*(L_2) \end{bmatrix} + \frac{z(1)}{L_{pkt}} \begin{bmatrix} r^*(1+L_1) \\ \vdots \\ r^*(1) \\ \vdots \\ r^*(1-L_2) \end{bmatrix} + \dots + \frac{z(L_{pkt}-1)}{L_{pkt}} \begin{bmatrix} r^*(L_{pkt}-1+L_1) \\ \vdots \\ r^*(L_{pkt}-1) \\ \vdots \\ r^*(L_{pkt}-1-L_2) \end{bmatrix} \quad (2.40)$$

shows a pattern in  $z(n)$  and  $\mathbf{r}(n)$ . The  $k$ th value of  $\nabla J$  is

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m) r^*(m-k), \quad -L_1 \leq k \leq L_2. \quad (2.41)$$

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

$$\rho(n) = r^*(n). \quad (2.42)$$

Now

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m) \rho(k-m). \quad (2.43)$$

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 long result of the convolution sum  $m(n)$  has support on  $-L_{\text{pkt}}+1 \leq n \leq L_{\text{pkt}}-1$ . Putting all the pieces together, we have

$$\begin{aligned} m(n) &= \sum_{m=0}^{L_{pkt}-1} z(m) \rho(n-m) \\ &= \sum_{m=0}^{L_{pkt}-1} z(m) r^*(m-n) \end{aligned} \quad (2.44)$$

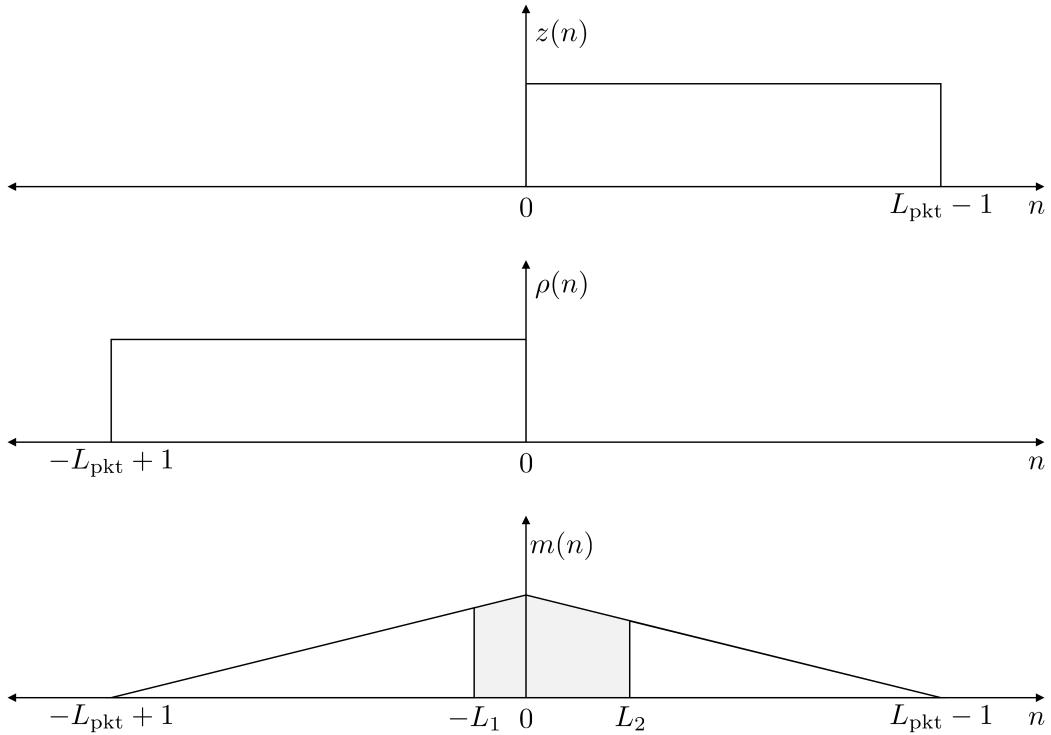


Figure 2.10: Diagram showing the relationships between  $z(n)$ ,  $\rho(n)$  and  $m(n)$ .

Comparing Equation (2.43) and (2.44) shows that

$$\nabla J(k) = \frac{1}{L_{\text{pkt}}} m(k), \quad -L_1 \leq k \leq L_2. \quad (2.45)$$

The values of interest are shown in Figure 2.10.

This suggest the following matlab code for computing computing the gradient vector  $\nabla J$  and implementing CMA.

Table 2.1: CMA

```
1 c_CMA = c_MMSE;
2 for i = 1:its
3     yy = conv(r,c_CMA);
4     y = yy(L1+1:end-L2); % trim yy
5     z = 2*(y.*conj(y)-1).*y;
6     Z = fft(z,Nfft);
7     R = fft(conj(r(end:-1:1)),Nfft)
8     m = ifft(Z.*R);
9     delJ = m(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
```

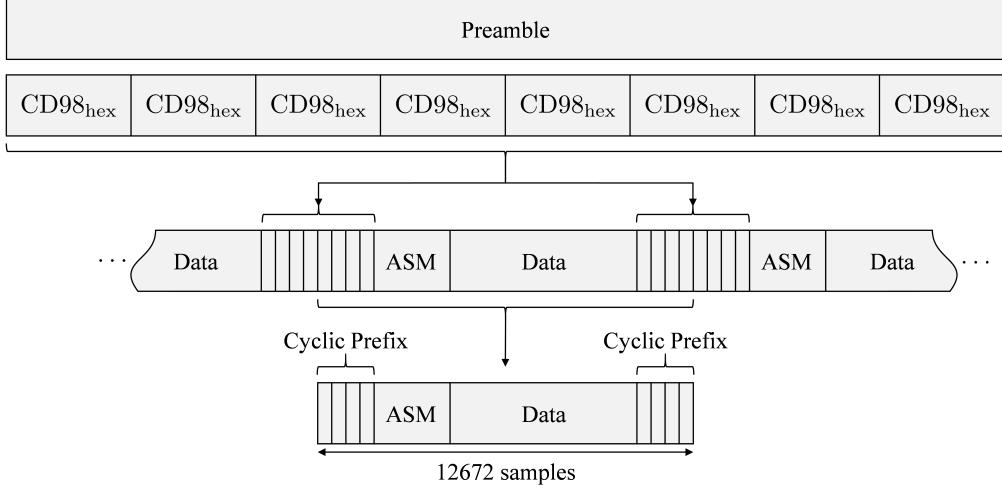


Figure 2.11: A diagram showing how the iNET packet is used as a cyclic prefix.

### 2.4.3 The Frequency Domain Equalizers

Frequency Domain Equalizer One (FDE1) and Frequency Domain Equalizer Two (FDE2) are very similar structure. FDE1 and FDE2 are adapted from Williams and Saquib [9, eq. (11) and (12)] [10]. The frequency domain equalizers are the frequency-domain equivalent to the MMSE equalizer.

Using circular convolution analysis, frequency-domain equalization performed on a packet structure must have cyclic prefix that is longer than the channel response [11–14]. Half of the iNET preamble is used as the cyclic prefix because it has eight repetitions of  $CD98_{hex}$ . Figure 2.11 shows how the iNET packet is used as a cyclic prefix. The length of the cyclic prefix is half the preamble length  $L_p/2 = 128$ . The length of the cyclic prefix is much longer than channel impulse response  $L_h = 38$ .

#### Frequency Domain Equalizer One

FDE1 is the MMSE equalizer applied in the frequency domain from [9, eq. (11)]:

$$C_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{\text{FFT}}} \text{ for } k = 0, 1, \dots, N_{\text{FFT}} - 1 \quad (2.46)$$

where  $N_{\text{FFT}} = 2^u = 16,384$  where  $u = \lceil \log_2(L_{\text{pkt}}) \rceil = 14$ . The term  $\lceil \log_2(L_{\text{pkt}}) \rceil$  is the ceiling of  $\log_2(L_{\text{pkt}})$ . The term  $C_{\text{FDE1}}(e^{j\omega_k})$  is the frequency response of FDE1 at  $\omega_k$ . The term  $\hat{H}(e^{j\omega_k})$  is the  $k$ th element of the length  $N_{\text{FFT}}$  of the channel estimate  $\hat{\mathbf{h}}$  frequency response at  $\omega_k$ . The term  $\hat{\sigma}^2$  is the noise variance estimate, this term is completely independent of frequency because the noise is assumed to be spectrally flat or white.

Equation (2.46) is straight forward to implement in GPUs. FDE1 is extremely fast and computationally efficient.

### Frequency Domain Equalizer Two

FDE2 is also the MMSE equalizer applied in the frequency domain. Unlike FDE1, FDE2 incorporates knowledge of the SOQPSK-TG power spectral density. The frequency response of FDE2 is [9, eq. (12)]

$$C_{\text{FDE2}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1 \quad (2.47)$$

where  $\Psi(e^{j\omega_k})$  is the power spectral density of SOQPSK-TG. Figure 2.12 shows  $\Psi(e^{j\omega_k})$  derived using welch's method of averaging periodograms based on length  $N_{\text{FFT}} = 16,384$  FFTs of SOQPSK-TG samples at a sample rate equivalent to 2 samples/bit. The term  $\Psi(e^{j\omega_k})$  eliminates out of band multipath that may be challenging to estimate and over come.

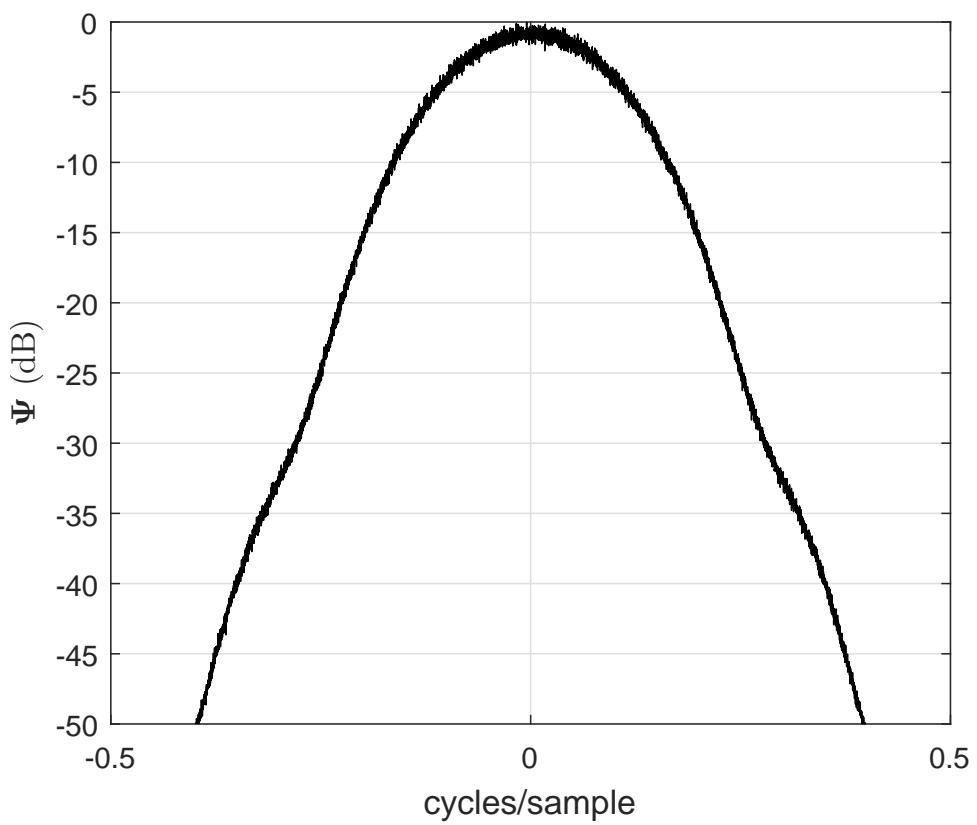


Figure 2.12: SOQPSK-TG power spectral density.



## 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. Tesla 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 are called. 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  $C_1$  is the sum of the vectors  $A_1$  and  $B_1$  computed in the CPU. The vector  $C_2$  is the sum of the vectors  $A_2$  and  $B_2$  computed in the GPU. Line 42 the CPU computes  $C_1$  by summing elements of  $A_1$  and  $B_1$  together *sequentially*. Figure 3.2 shows how the

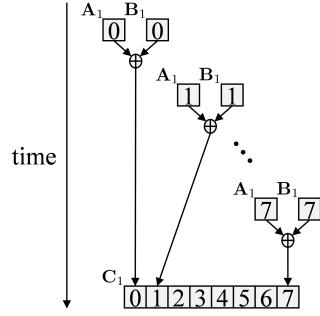


Figure 3.2: A block diagram of how a CPU sequentially performs vector addition.

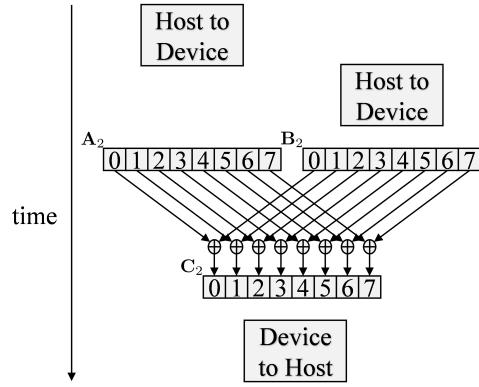


Figure 3.3: A block diagram of how a GPU performs vector addition in parallel.

CPU sequentially 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 verse GPU code.

```

1 #include <iostream>
2 #include <stdlib.h>
3 #include <math.h>
4 using namespace std;
5
6 void VecAddCPU(float* destination, float* source0, float* source1, int myLength) {
7     for(int i = 0; i < myLength; i++)
8         destination[i] = source0[i] + source1[i];
9 }
10
11 __global__ void VecAddGPU(float* destination, float* source0, float* source1, int lastThread) {
12     int i = blockIdx.x*blockDim.x + threadIdx.x;
13
14     // don't access elements out of bounds
15     if(i >= lastThread)
16         return;
17
18     destination[i] = source0[i] + source1[i];
19 }
20
21 int main(){
22     int N = pow(2,22);
23     cout << N << endl;
24     /**
25      * Vector Addition on CPU
26      */
27     // allocate memory on host
28     float *A1;
29     float *B1;
30     float *C1;
31     A1 = (float*) malloc (N*sizeof(float));
32     B1 = (float*) malloc (N*sizeof(float));
33     C1 = (float*) malloc (N*sizeof(float));
34
35     // Initialize vectors 0-99
36     for(int i = 0; i < N; i++){
37         A1[i] = rand()%100;
38         B1[i] = rand()%100;
39     }
40
41     // vector sum C1 = A1 + B1
42     VecAddCPU(C1, A1, B1, N);
43
44     /**
45      * Vector Addition on GPU
46      */
47     // allocate memory on host for result
48     float *C2;
49     C2 = (float*) malloc (N*sizeof(float));
50
51     // allocate memory on device for computation
52     float *A2_gpu;
53     float *B2_gpu;
54     float *C2_gpu;
55     cudaMalloc(&A2_gpu, sizeof(float)*N);
56     cudaMalloc(&B2_gpu, sizeof(float)*N);
57     cudaMalloc(&C2_gpu, sizeof(float)*N);
58
59     // Copy vectors A and B from host to device
60     cudaMemcpy(A2_gpu, A1, sizeof(float)*N, cudaMemcpyHostToDevice);
61     cudaMemcpy(B2_gpu, B1, sizeof(float)*N, cudaMemcpyHostToDevice);
62
63     // Set optimal number of threads per block
64     int T_B = 32;
65
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 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

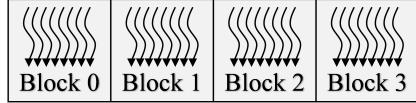


Figure 3.4: 32 threads launched in 4 thread blocks with 8 threads per block.

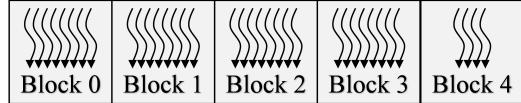


Figure 3.5: 36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.

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. Note that 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 the 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, L1 cache and shared memory. Local memory is fast but only kilobytes are available. The slowest memory is public memory in the form of L2 cache and global memory. Public memory is slow but gigabytes

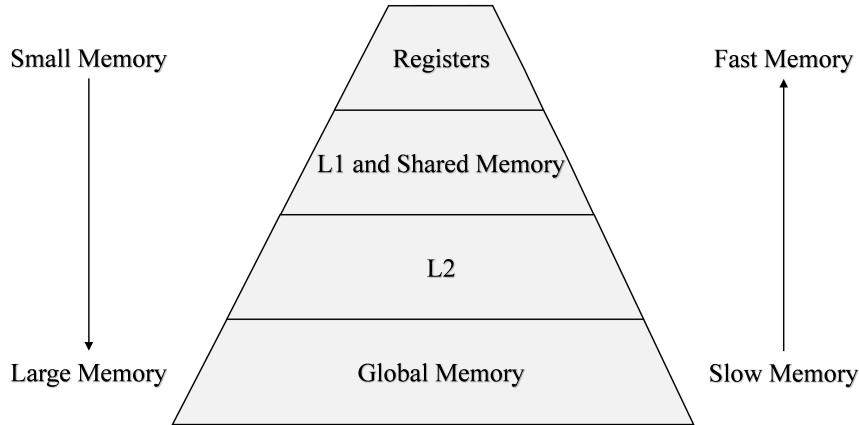


Figure 3.6: Diagram comparing memory size and speed. Global memory is massive but extremely slow. Registers are extremely fast but there are very few.

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 L2 cache or global memory are physically located *off* the GPU chip. The dashed box shows that registers, L1 cache or shared memory are physically located *on* the GPU chip. A L2 cache or global memory access takes over 60 clock cycles because the memory is off chip. A registers, L1 cache or shared 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 L1 cache. Threads in a block can coordinate using shared memory because shared memory private to the thread block. All threads have access to 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 Tesla 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 will have very little memory resources available if a GPU kernel launches 1024 threads

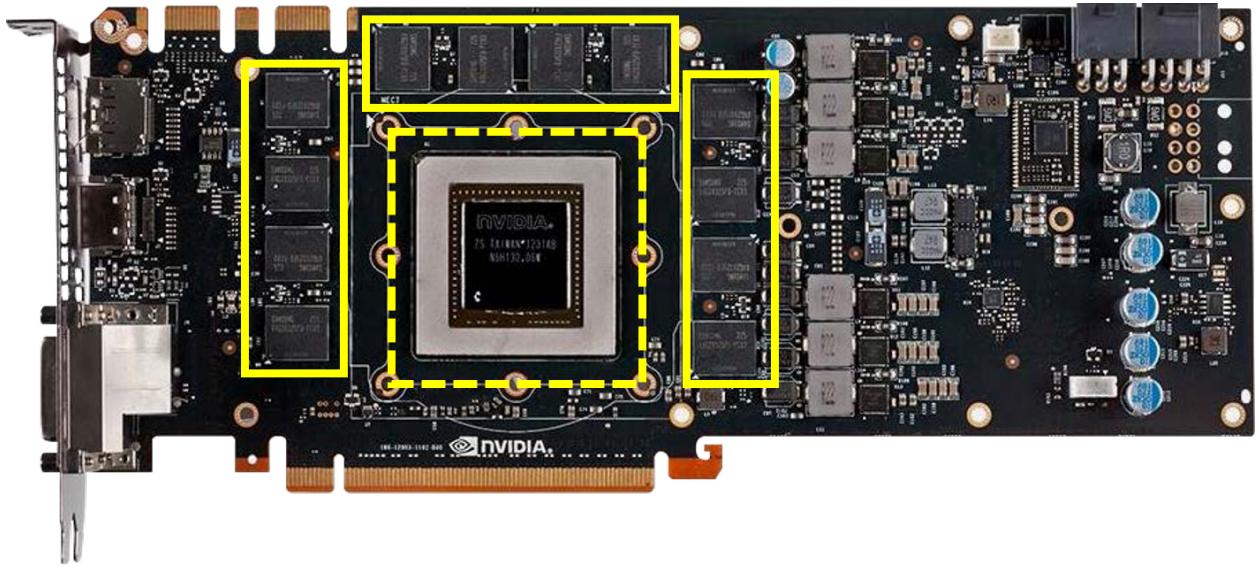


Figure 3.7: Example of an NVIDIA GPU card. The GPU chip with registers, L1 cache and shared memory is shown in the dashed box. The L2 cache and global memory is shown off chip in the solid boxes.

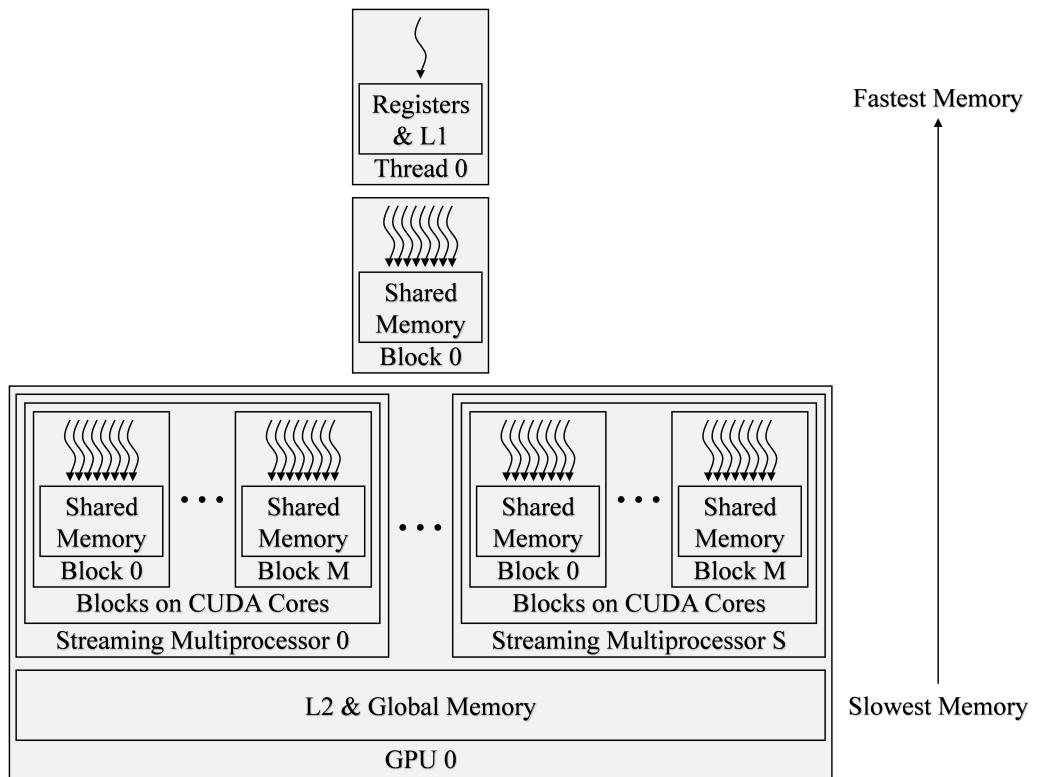


Figure 3.8: A block diagram where local, shared, and global memory is located. Each thread has private local memory. Each thread block has private shared memory. The GPU has global memory that all threads can access.

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

per block. Threads will have a lot of memory resources available if a GPU kernel launches 32 threads per block. This section will show that finding the optimum number of threads per block can drastically 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. 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 2.

Adjusting the number of threads per block does not always drastically 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.

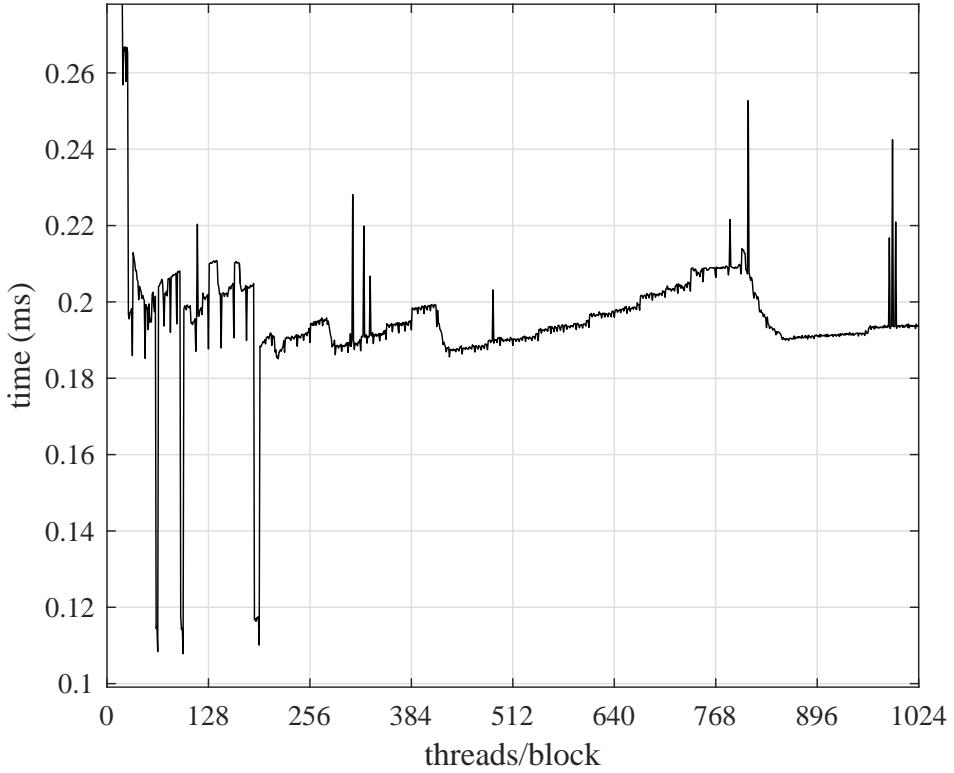


Figure 3.9: Plot showing how execution time is affected by changing the number of threads per block. The optimal execution time for an example GPU kernel is 0.1078 ms at the optimal 96 threads per block.

### 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 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

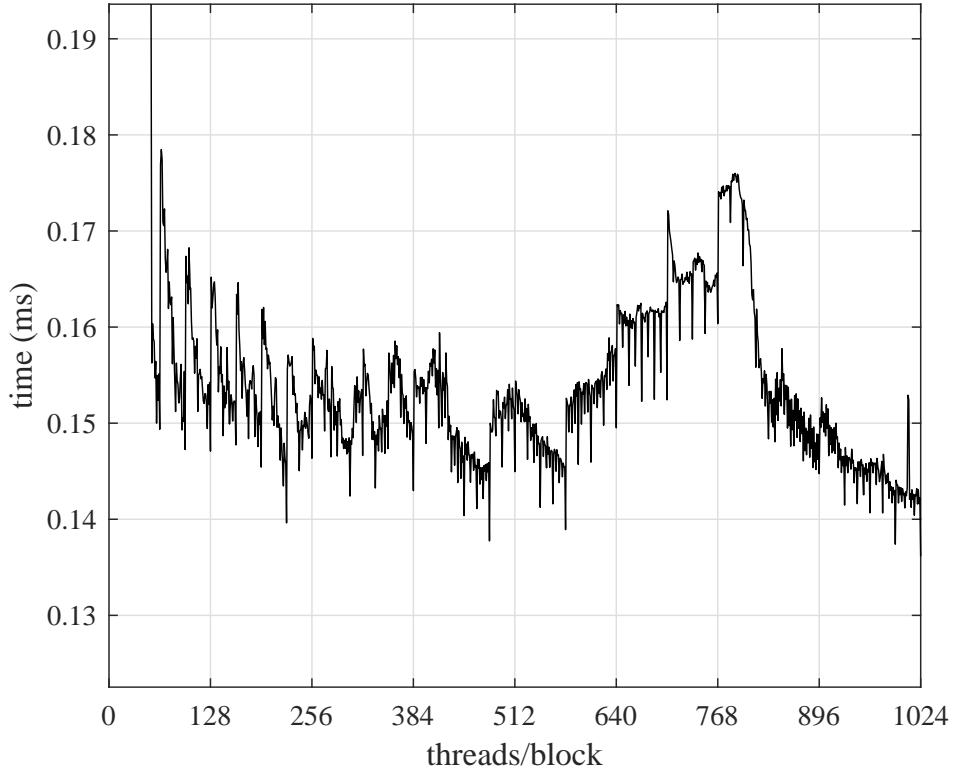


Figure 3.10: Plot showing the number of threads per block doesn't always drastically affect execution time.

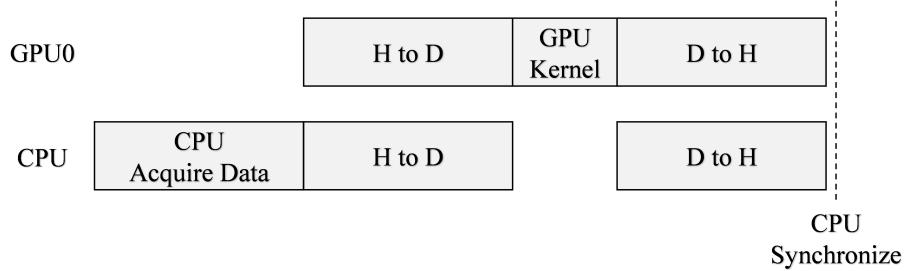


Figure 3.11: The typical approach of CPU and GPU operations. This block diagram shows the profile of Listing 3.3.

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.

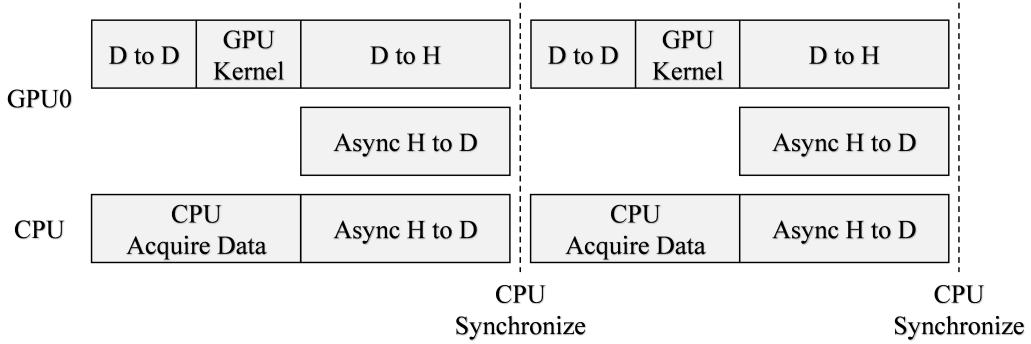


Figure 3.12: GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 3.4.

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 temporary 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.

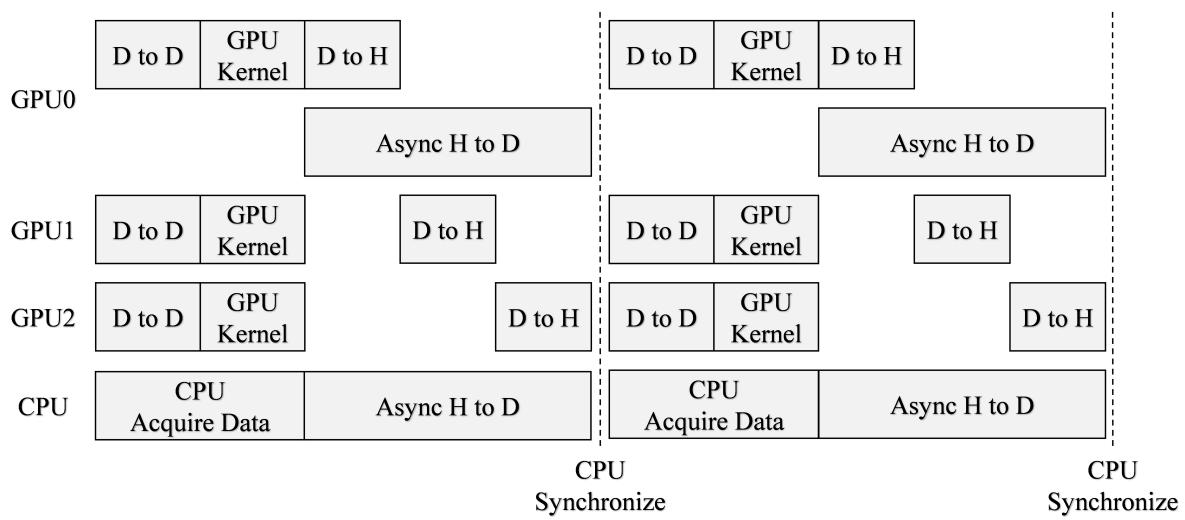


Figure 3.13: A block diagram of pipelining a CPU with three GPUs.

**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.

```

1 int main()
2 {
3     ...
4     // CPU Acuire Data
5     myADC.acquire(vec);
6
7     // Launch instructions on GPU
8     cudaMemcpy(dev_vec0, vec,      numBytes, cudaMemcpyHostToDevice);
9     GPUkernel<<<1, N>>>(dev_vec0);
10    cudaMemcpy(vec,      dev_vec0, numBytes, cudaMemcpyDeviceToHost);
11
12    // Synchronize CPU with GPU
13    cudaDeviceSynchronize();
14    ...
15 }
```

**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.

```

1 int main()
2 {
3     ...
4     // Launch instructions on GPU
5     cudaMemcpy(dev_vec, dev_temp, numBytes, cudaMemcpyDeviceToDevice);
6     GPUkernel<<<N, M>>>(dev_vec);
7     cudaMemcpy(vec,      dev_vec, numBytes, cudaMemcpyDeviceToHost);
8
9     // CPU Acuire Data
10    myADC.acquire(vec);
11    cudaMemcpyAsync(dev_temp, vec, numBytes, cudaMemcpyHostToDevice);
12
13    // Synchronize CPU with GPU
14    cudaDeviceSynchronize();
15    ...
16
17    ...
18    // Launch instructions on GPU
19    cudaMemcpy(dev_vec, dev_temp, numBytes, cudaMemcpyDeviceToDevice);
20    GPUkernel<<<N, M>>>(dev_vec);
21    cudaMemcpy(vec,      dev_vec, numBytes, cudaMemcpyDeviceToHost);
22
23    // CPU Acuire Data
24    myADC.acquire(vec);
25    cudaMemcpyAsync(dev_temp, vec, numBytes, cudaMemcpyHostToDevice);
26
27    // Synchronize CPU with GPU
28    cudaDeviceSynchronize();
29    ...
30 }
```

### 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 be reduced by 260 ms. This section will use the following notation:

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

Discrete time convolution applies the filter  $\mathbf{h}$  to the signal  $\mathbf{x}$  resulting in the filter signal  $\mathbf{y}$ .

Convolution in the time domain is

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

and the frequency domain is

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

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.

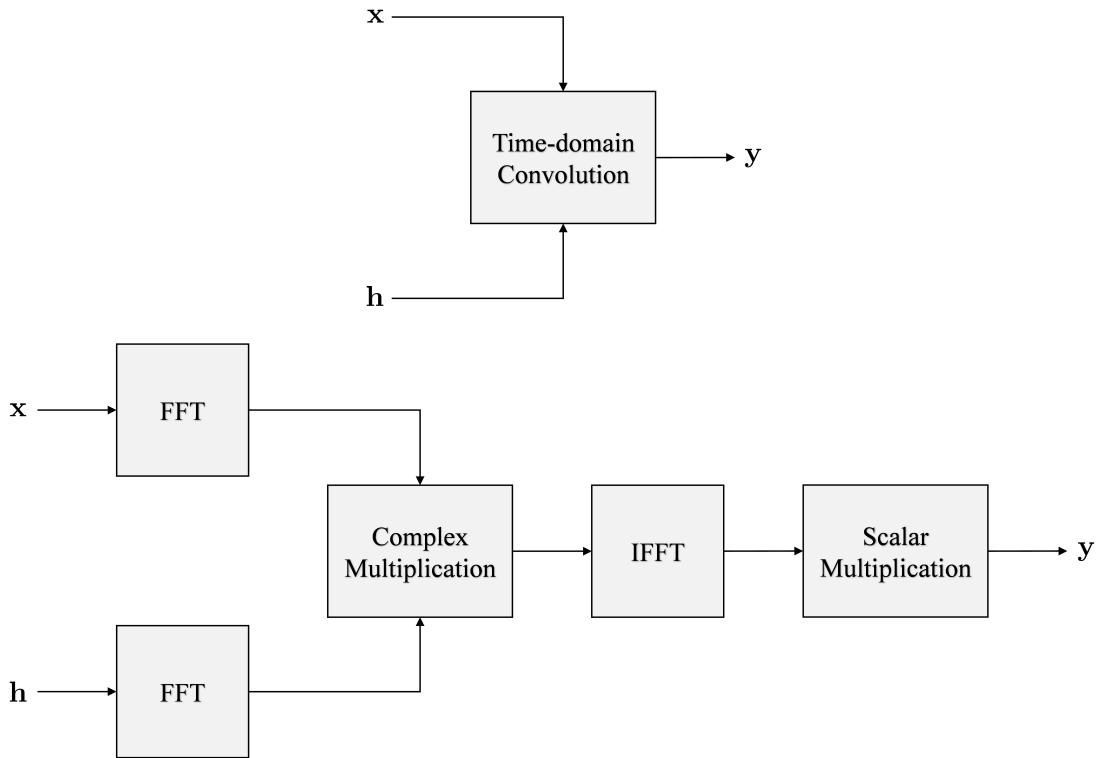


Figure 3.14: Block diagrams showing time-domain convolution and frequency-domain convolution.

- 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

$$(A + jB) \times (C + jD) = (AC - BD) + j(AD + BC) \quad (3.3)$$

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

$$8LC \text{ flops} \quad (3.4)$$

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

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

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

Sections 2.1 and 2.3.5 show the PAQ system has one signal length,  $N = L_{\text{pkt}} = 12672$  samples and two filter lengths  $L = L_{\text{df}} = 21$  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 12672 sample signal requires less flops in the frequency domain and applying a 21 tap filter to a 12672 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

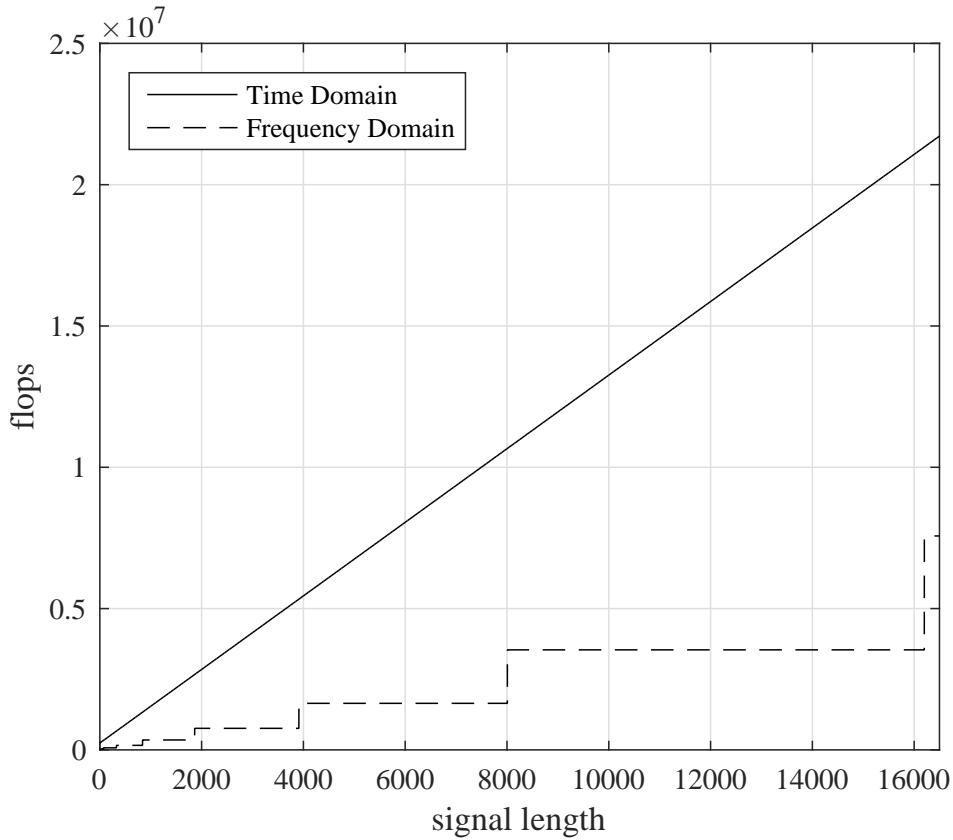


Figure 3.15: Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 186 tap complex filter.

time of the code in Listing 3.5 (at the end of the chapter) was measured. The code implements convolution five different ways:

- 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
- 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

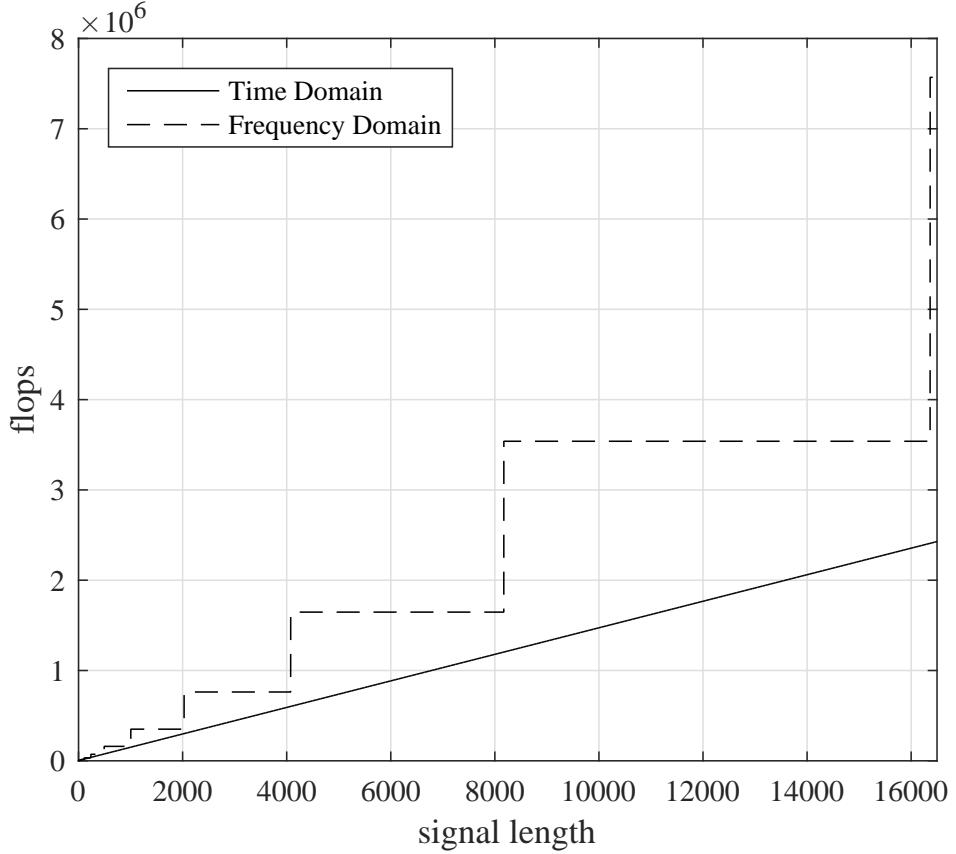


Figure 3.16: Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 21 tap complex filter.

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.

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 visa versa. Sub-windows emphasize points that are of interest to the PAQ system. The variations in the time-domain CPU execution times are due to the claims on the host CPU resources by the operating

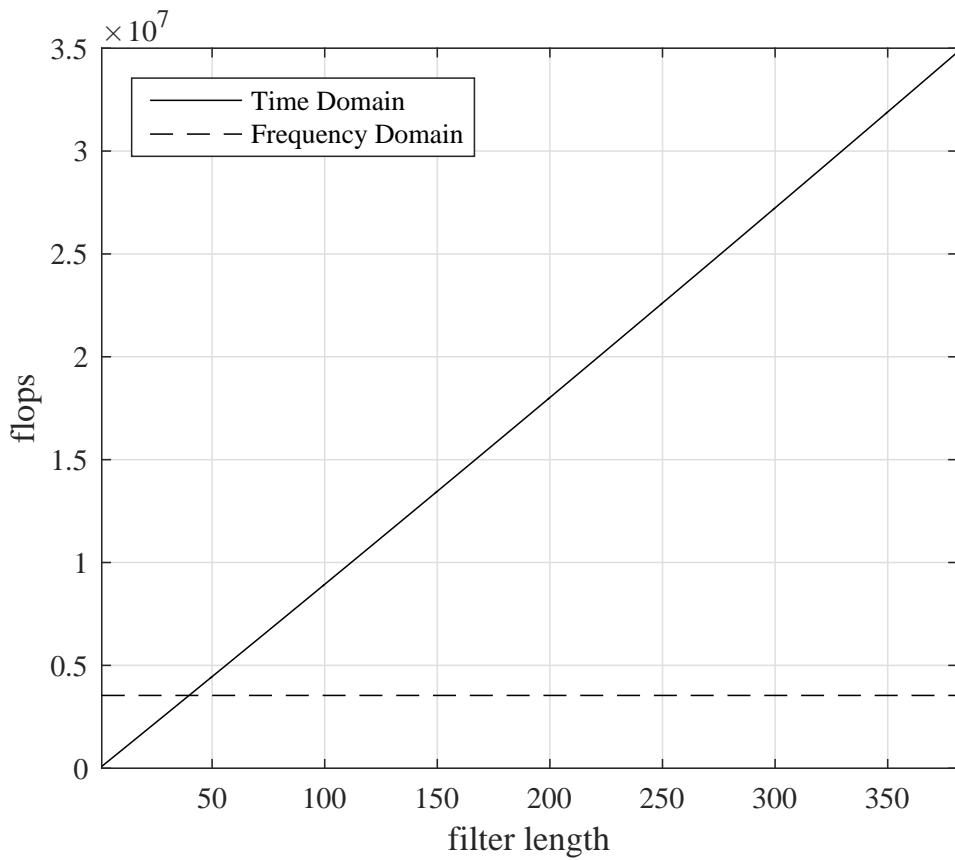


Figure 3.17: Comparison of number of floating point operations (flops) required to convolve a 12672 sample complex signal with a variable length tap complex filter.

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	ConvGPUsshared	282	293
GPU frequency domain	cuffFT	301	327

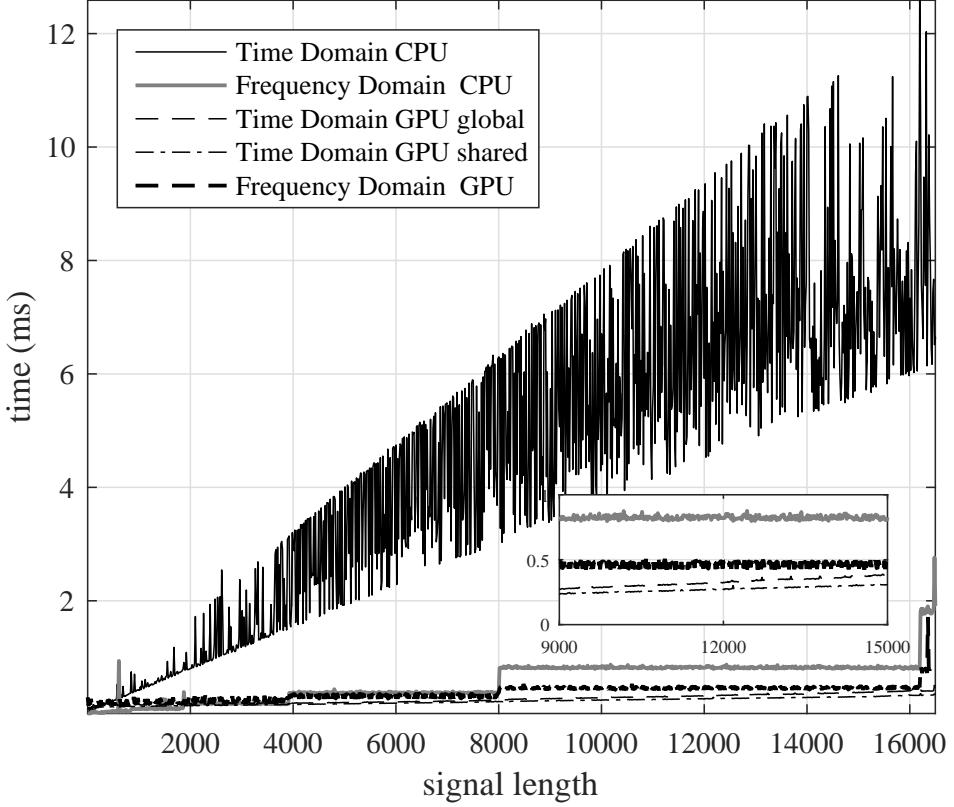


Figure 3.18: Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 186 taps. The comparison is messy with out lower bounding.

system. To clean up the time samples, local minimums 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.

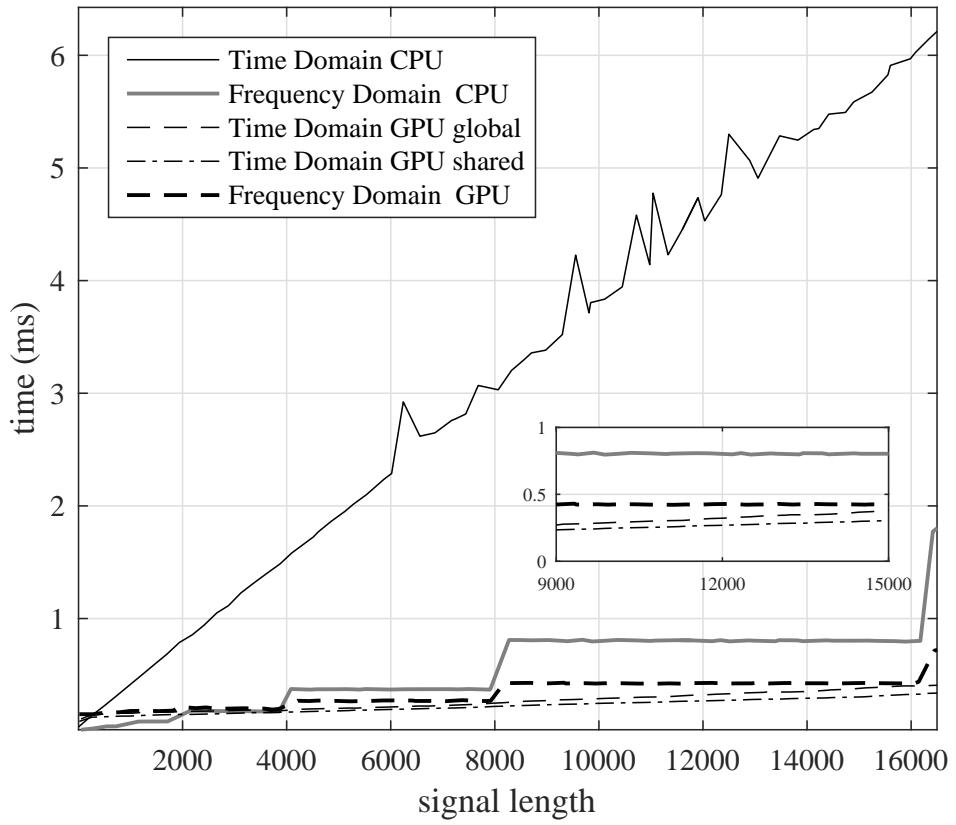


Figure 3.19: Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 186 taps. A lower bound was applied by searching for a local minimum in 15 sample width windows.

Table 3.3: Convolution computation times with signal length 12672 and filter length 186 on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
CPU time domain	ConvCPU	5.3000
CPU frequency domain	FFTW	0.7972
GPU time domain global	ConvGPU	0.3321
GPU time domain shared	ConvGPUshared	0.2748
GPU frequency domain	cuFFT	0.4224

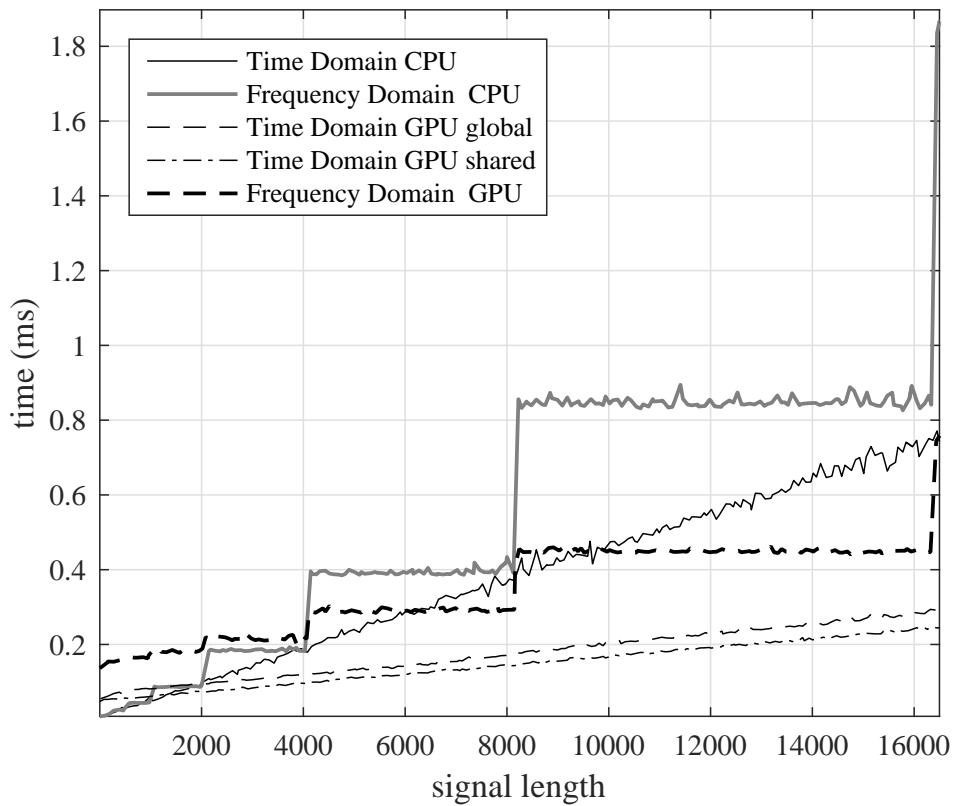


Figure 3.20: Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 21 taps. A lower bound was applied by searching for a local minimum in 5 sample width windows.

Table 3.4: Convolution computation times with signal length 12672 and filter length 21 on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
CPU time domain	ConvCPU	0.5878
CPU frequency domain	FFTW	0.8417
GPU time domain global	ConvGPU	0.4476
GPU time domain shared	ConvGPUsShared	0.1971
GPU frequency domain	cuFFT	0.3360

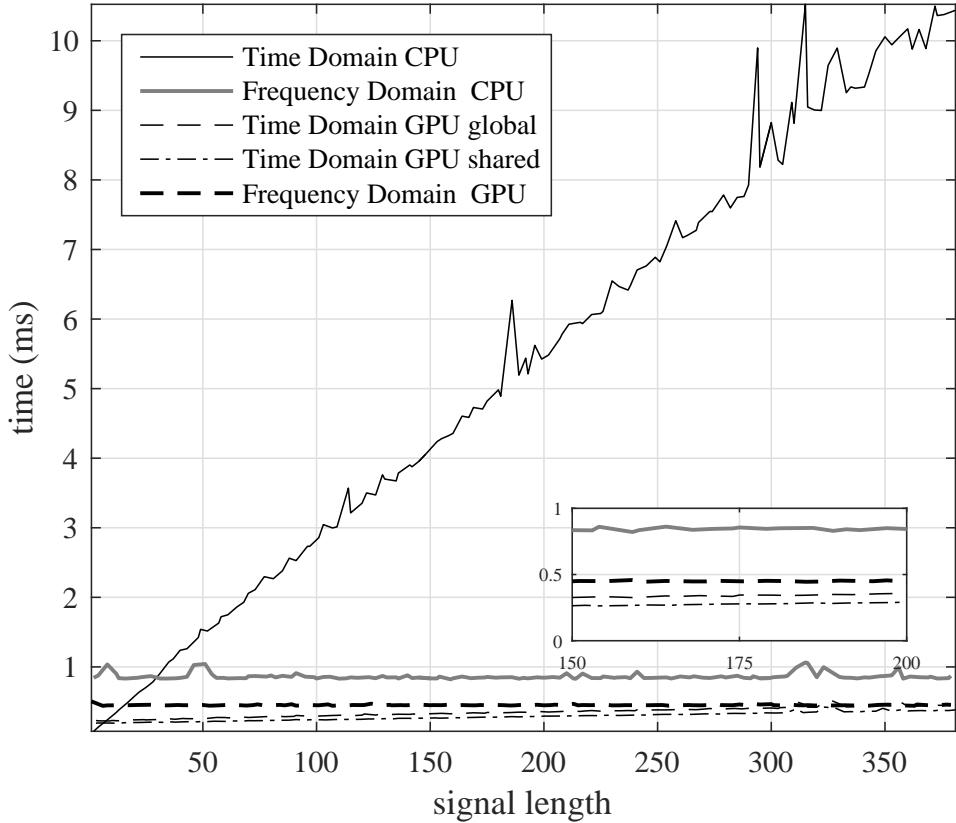


Figure 3.21: Comparison of a complex convolution on CPU verse GPU. The filter length is variable and the signal is fixed at 12672 samples. A lower bound was applied by searching for a local minimums in 3 sample width windows.

### 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 libaries achieve more Gflops than calling GPU kernels multiple times. Listing 3.6 (at the end of the chapter) shows three GPU

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

implementations of convolution using batch processing and Table 3.5 shows how the execution time of the code was measured.

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.

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 visa 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 21 tap filters.

Until now, convolving one signal with only one filter has been considered. Figure 2.7 showed the received signal is filtered by two cascaded filters: an equalizer filter and a detection

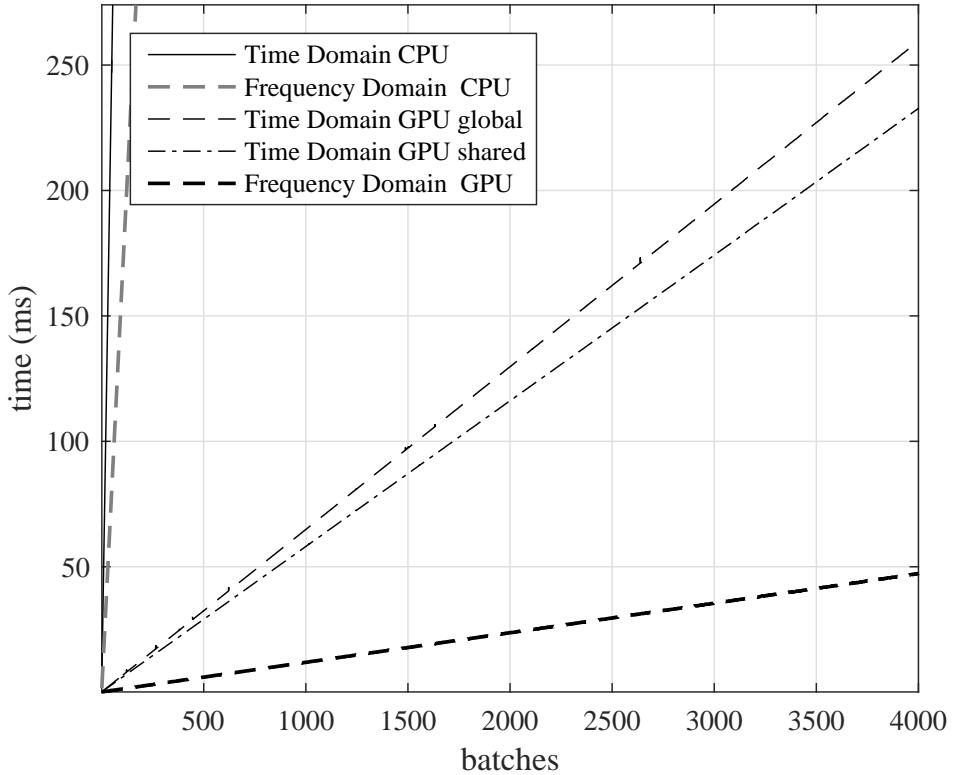


Figure 3.22: Comparison of a batched complex convolution on a CPU and GPU. The number of batches is variable while the signal and filter length is set to 12672 and 186.

Table 3.6: Convolution using batch processing execution times with for a 12672 sample signal and 186 tap filter on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
GPU time domain global	ConvGPU	201.29
GPU time domain shared	ConvGPUsShared	180.272
GPU frequency domain	cuFFT	36.798

Table 3.7: Convolution using batch processing execution times with for a 12672 sample signal and 21 tap filter on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
GPU time domain global	ConvGPU	27.642
GPU time domain shared	ConvGPUsShared	20.4287
GPU frequency domain	cuFFT	36.7604

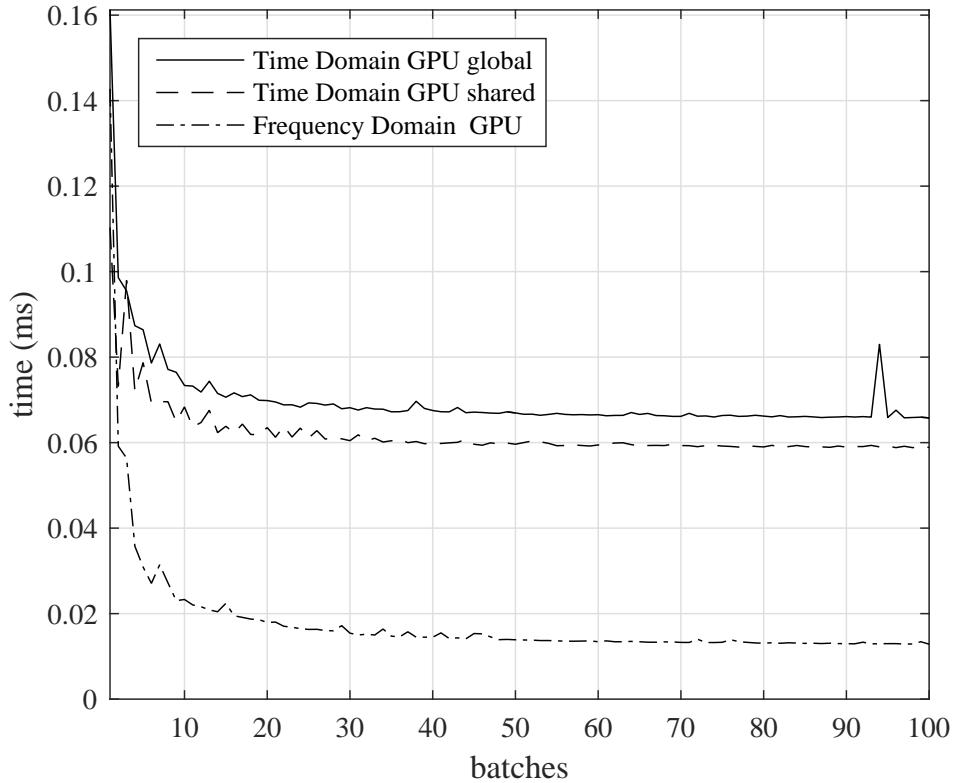


Figure 3.23: Comparison on execution time per batch for complex convolution. The number of batches is variable while the signal and filter length is set to 12672 and 186.

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 21 tap detection filter. The second time-domain convolution applies the composite  $206 = 186 + 21 -$  tap filter to a 12672 sample signal. Table 3.8 shows the execution times for the signal length and filter lengths of the PAQ system when performing cascaded convolution using batch processing. Cascaded-convolution using batch processing in the frequency domain is fastest.

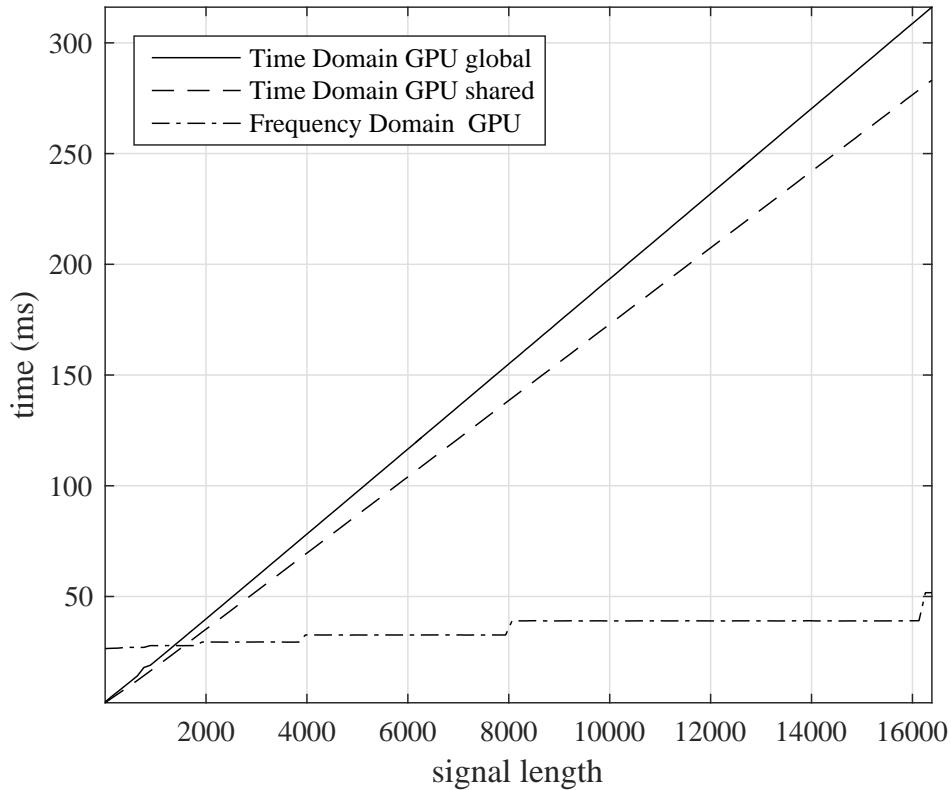


Figure 3.24: Comparison of complex convolution using batch processing on a GPU. The signal length is variable and the filter is fixed at 186 taps.

Table 3.8: Batched convolution execution times with for a 12672 sample signal and cascaded 21 and 186 tap filter on a Tesla K40c GPU.

Algorithm	Function or Library	Execution Time (ms)
GPU time domain global	ConvGPU	223.307
GPU time domain shared	ConvGPUsShared	200.018
GPU frequency domain	cuffFT	39.0769

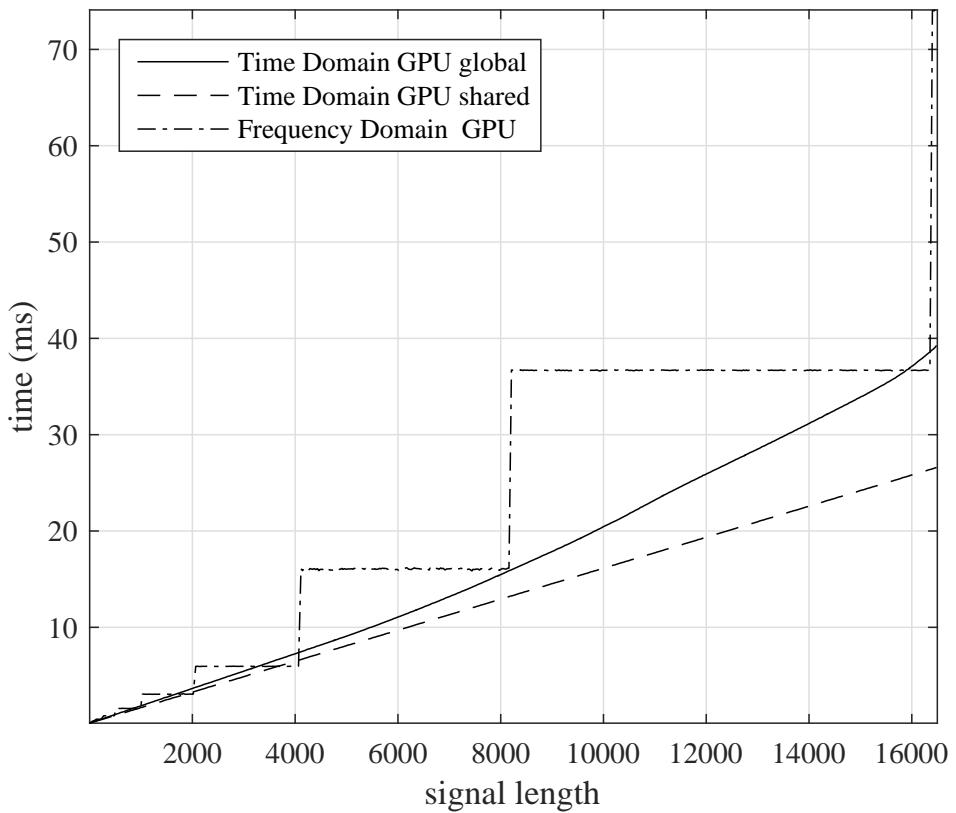


Figure 3.25: Comparison of complex convolution using batch processing on a GPU. The signal length is variable and the filter is fixed at 21 taps.

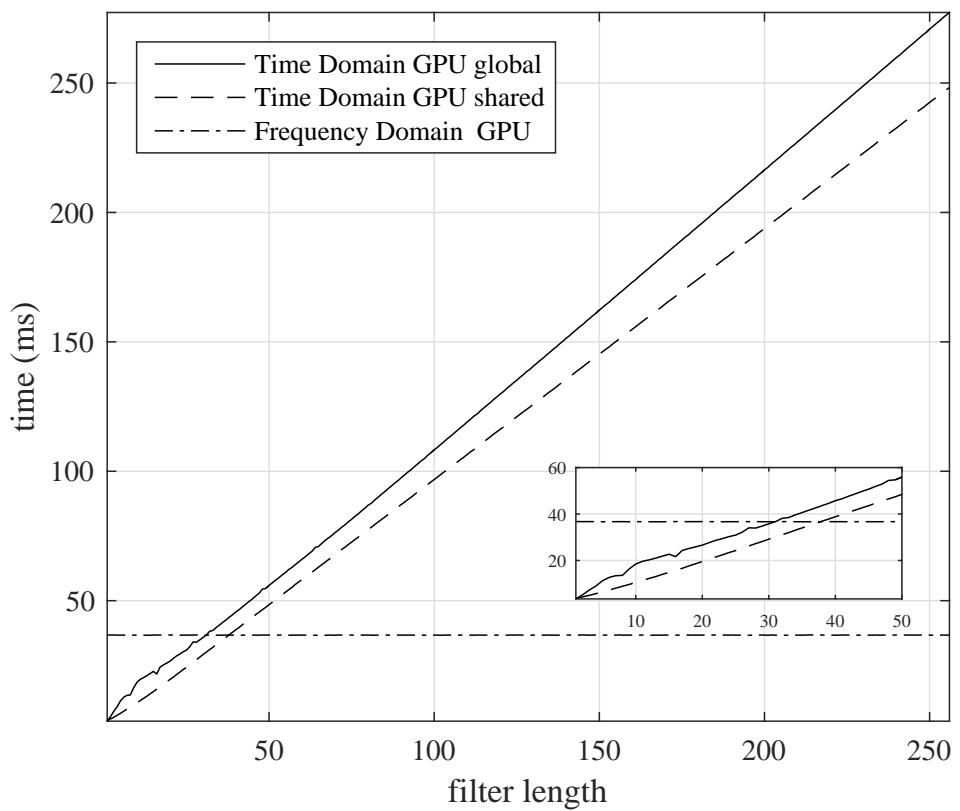


Figure 3.26: Comparison of complex convolution using batch processing on a GPU. The filter length is variable and the signal length is set to 12672 samples.

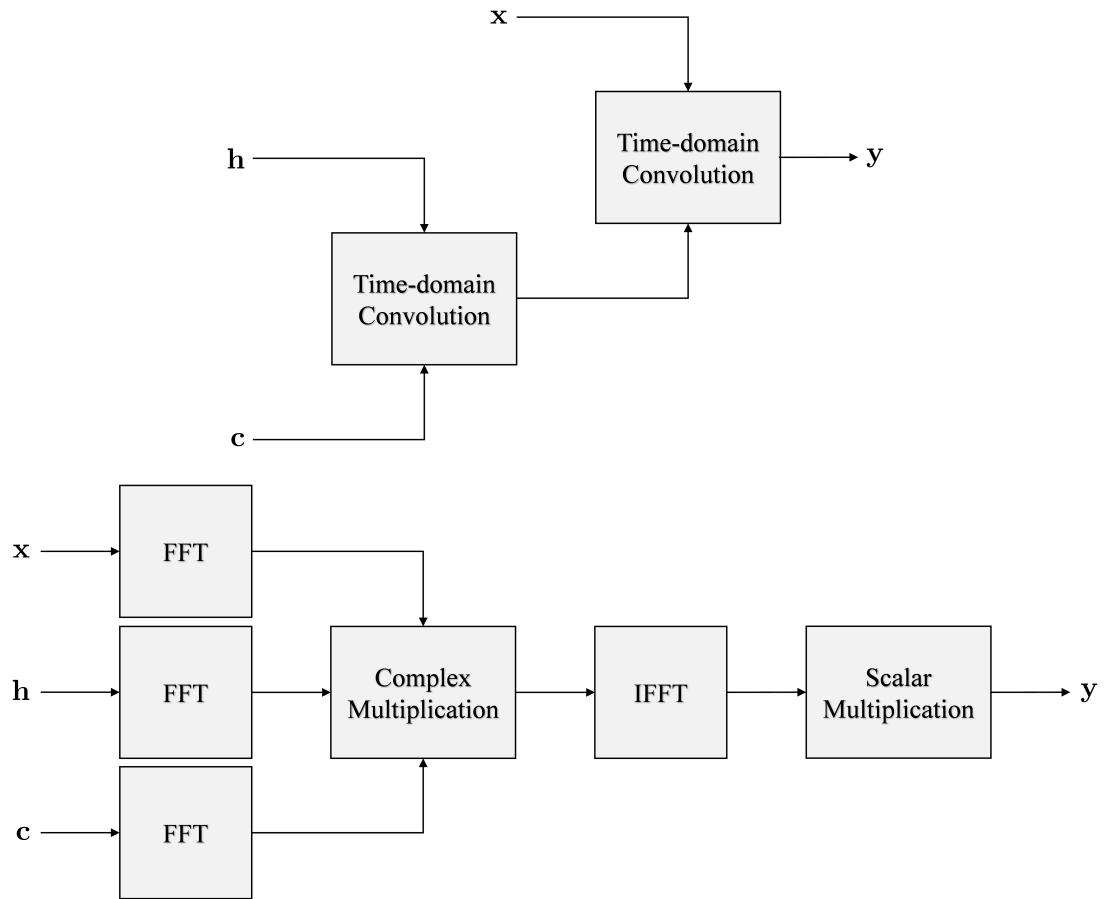


Figure 3.27: Block diagrams showing cascaded time-domain convolution and frequency-domain convolution.

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.

```

1 #include <iostream>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <cufft.h>
5 #include <fstream>
6 #include <string>
7 #include <fftw3.h>
8 using namespace std;
9
10
11 void ConvCPU(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
12     for(int yIdx = 0; yIdx < Lx+Lh-1; yIdx++) {
13         cufftComplex temp;
14         temp.x = 0;
15         temp.y = 0;
16         for(int hIdx = 0; hIdx < Lh; hIdx++) {
17             int xAccessIdx = yIdx-hIdx;
18             if(xAccessIdx>=0 && xAccessIdx<Lx) {
19                 // temp += x[xAccessIdx]*h[hIdx];
20                 float A = x[xAccessIdx].x;
21                 float B = x[xAccessIdx].y;
22                 float C = h[hIdx].x;
23                 float D = h[hIdx].y;
24                 cufftComplex result;
25                 result.x = A*C-B*D;
26                 result.y = A*D+B*C;
27                 temp.x += result.x;
28                 temp.y += result.y;
29             }
30         }
31         y[yIdx] = temp;
32     }
33 }
34 }
35
36 __global__ void ConvGPU(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
37     int yIdx = blockIdx.x*blockDim.x + threadIdx.x;
38
39     int lastThread = Lx+Lh-1;
40
41     // don't access elements out of bounds
42     if(yIdx >= lastThread)
43         return;
44
45     cufftComplex temp;
46     temp.x = 0;
47     temp.y = 0;
48     for(int hIdx = 0; hIdx < Lh; hIdx++) {
49         int xAccessIdx = yIdx-hIdx;
50         if(xAccessIdx>=0 && xAccessIdx<Lx) {
51             // temp += x[xAccessIdx]*h[hIdx];
52             float A = x[xAccessIdx].x;
53             float B = x[xAccessIdx].y;
54             float C = h[hIdx].x;
55             float D = h[hIdx].y;
56             cufftComplex result;
57             result.x = A*C-B*D;
58             result.y = A*D+B*C;
59             temp.x += result.x;
60             temp.y += result.y;
61         }
62     }
63     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,      (
215         fftwf_complex*)mySignal2_fft,           FFTW_FORWARD, FFTW_MEASURE);
216     fftwf_plan forwardPlanFilter = fftwf_plan_dft_1d(M, (fftwf_complex*)myFilter2,      (
217         fftwf_complex*)myFilter2_fft,           FFTW_FORWARD, FFTW_MEASURE);
218     fftwf_plan backwardPlanConv = fftwf_plan_dft_1d(M, (fftwf_complex*)mySignal2_fft,      (
219         fftwf_complex*)myConv2_timeReversed, FFTW_FORWARD, FFTW_MEASURE);
220
221     cufftComplex zero; zero.x = 0; zero.y = 0;
222     for(int i = 0; i < M; i++){
223         if(i<N)
224             mySignal2[i] = mySignal1[i];
225         else
226             mySignal2[i] = zero;
227
228         if(i<L)
229             myFilter2[i] = myFilter1[i];
230         else
231             myFilter2[i] = zero;
232     }
233
234     fftwf_execute(forwardPlanSignal);
235     fftwf_execute(forwardPlanFilter);
236
237     for (int i = 0; i < M; i++){
238         // mySignal2_fft = mySignal2_fft*myFilter2_fft;
239         float A = mySignal2_fft[i].x;
240         float B = mySignal2_fft[i].y;
241         float C = myFilter2_fft[i].x;
242         float D = myFilter2_fft[i].y;
243         cufftComplex result;
244         result.x = A*C-B*D;
245         result.y = A*D+B*C;
246         mySignal2_fft[i] = result;
247     }
248
249     fftwf_execute(backwardPlanConv);
250
251     // myConv2 from fftwf must be time reversed and scaled
252     // to match Matlab, myConv1, myConv3, myConv4 and myConv5
253     cufftComplex result;
254     for (int i = 0; i < M; i++){
255         result.x = myConv2_timeReversed[M-i].x/M;
256         result.y = myConv2_timeReversed[M-i].y/M;
257         myConv2[i] = result;
258     }
259     result.x = myConv2_timeReversed[0].x/M;
260     result.y = myConv2_timeReversed[0].y/M;
261     myConv2[0] = result;
262
263     fftwf_destroy_plan(forwardPlanSignal);
264     fftwf_destroy_plan(forwardPlanFilter);
265     fftwf_destroy_plan(backwardPlanConv);

```

```

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      * Time-domain Convolution GPU Using Shared Memory
281      */
282     cudaMemcpy(dev_mySignal4, mySignal1, sizeof(cufftComplex)*N, cudaMemcpyHostToDevice);
283     cudaMemcpy(dev_myFilter4, myFilter1, sizeof(cufftComplex)*L, cudaMemcpyHostToDevice);
284
285     T_B = 512;
286     B = C/T_B;
287     if(C % T_B > 0)
288         B++;
289     ConvGPUshared<<<B, T_B,L*sizeof(cufftComplex)>>>(dev_myConv4, dev_mySignal4,
290             dev_myFilter4, N, L);
291
292     cudaMemcpy(myConv4, dev_myConv4, C*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
293
294     /**
295      * Frequency-domain Convolution GPU
296      */
297     cufftHandle plan;
298     int n[1] = {M};
299     cufftPlanMany(&plan, 1,n,NULL,1,1,NULL,1,1,CUFFT_C2C,1);
300
301     cudaMemset(dev_mySignal5, 0, M*sizeof(cufftComplex));
302     cudaMemset(dev_myFilter5, 0, M*sizeof(cufftComplex));
303
304     cudaMemcpy(dev_mySignal5, mySignal2, M*sizeof(cufftComplex), cudaMemcpyHostToDevice);
305     cudaMemcpy(dev_myFilter5, myFilter2, M*sizeof(cufftComplex), cudaMemcpyHostToDevice);
306
307     cufftExecC2C(plan, dev_mySignal5, dev_mySignal5, CUFFT_FORWARD);
308     cufftExecC2C(plan, dev_myFilter5, dev_myFilter5, CUFFT_FORWARD);
309
310     T_B = 512;
311     B = M/T_B;
312     if(M % T_B > 0)
313         B++;
314     PointToPointMultiply<<<B, T_B>>>(dev_mySignal5, dev_myFilter5, M);
315
316     cufftExecC2C(plan, dev_mySignal5, dev_mySignal5, CUFFT_INVERSE);
317
318     T_B = 128;
319     B = M/T_B;
320     if(M % T_B > 0)
321         B++;
322     float scalar = 1.0/((float)M);
323     ScalarMultiply<<<B, T_B>>>(dev_mySignal5, scalar, M);
324
325     cudaMemcpy(myConv5, dev_mySignal5, M*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
326
327     cufftDestroy(plan);
328
329     free(mySignal1);
330     free(mySignal2);

```

```
332     free(myFilter1);
333     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.

```

1 #include <cuda.h>
2 #include <iostream>
3 using namespace std;
4
5 __global__ void ConvGPU(cufftComplex* y_out, cufftComplex* x_in, cufftComplex* h_in, int Lx, int Lh,
6     int maxThreads) {
7     int threadNum = blockIdx.x*blockDim.x + threadIdx.x;
8     int convLength = Lx+Lh-1;
9
10    // Don't access elements out of bounds
11    if(threadNum >= maxThreads)
12        return;
13
14    int batch = threadNum/convLength;
15    int yIdx = threadNum%convLength;
16    cufftComplex* x = &x_in[Lx*batch];
17    cufftComplex* h = &h_in[Lh*batch];
18    cufftComplex* y = &y_out[convLength*batch];
19
20    cufftComplex temp;
21    temp.x = 0;
22    temp.y = 0;
23    for(int hIdx = 0; hIdx < Lh; hIdx++) {
24        int xAccessIdx = yIdx-hIdx;
25        if(xAccessIdx>=0 && xAccessIdx<Lx) {
26            // temp += x[xAccessIdx]*h[hIdx];
27            // (A+jB) (C+jD) = (AC-BD) + j(AD+BC)
28            float A = x[xAccessIdx].x;
29            float B = x[xAccessIdx].y;
30            float C = h[hIdx].x;
31            float D = h[hIdx].y;
32            cufftComplex complexMult;
33            complexMult.x = A*C-B*D;
34            complexMult.y = A*D+B*C;
35
36            temp.x += complexMult.x;
37            temp.y += complexMult.y;
38        }
39        y[yIdx] = temp;
40    }
41
42 __global__ void ConvGPUshared(cufftComplex* y_out, cufftComplex* x_in, cufftComplex* h_in, int Lx,
43     int Lh, int maxThreads) {
44
45     int threadNum = blockIdx.x*blockDim.x + threadIdx.x;
46     int convLength = Lx+Lh-1;
47     // Don't access elements out of bounds
48     if(threadNum >= maxThreads)
49         return;
50
51     int batch = threadNum/convLength;
52     int yIdx = threadNum%convLength;
53     cufftComplex* x = &x_in[Lx*batch];
54     cufftComplex* h = &h_in[Lh*batch];
55     cufftComplex* y = &y_out[convLength*batch];
56
57     extern __shared__ cufftComplex h_shared[];
58     if(threadIdx.x < Lh)
59         h_shared[threadIdx.x] = h[threadIdx.x];
60
61     __syncthreads();
62
63     cufftComplex temp;
64     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<Ly) {
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,
196               cudaMemcpyHostToDevice);
197     cudaMemcpy(dev_myFilter1, myFilter1, numBatches*sizeof(cufftComplex)*L,
198               cudaMemcpyHostToDevice);

```

```

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),
206                cudaMemcpyDeviceToHost);
207
208     /**
209      * Time-domain Convolution GPU Using Shared Memory
210      */
211     cudaMemcpy(dev_mySignal2, mySignal1, numBatches*sizeof(cufftComplex)*N,
212                cudaMemcpyHostToDevice);
213     cudaMemcpy(dev_myFilter2, myFilter1, numBatches*sizeof(cufftComplex)*L,
214                cudaMemcpyHostToDevice);
215
216     maxThreads = C*numBatches;
217     T_B = 256;
218     B = maxThreads/T_B;
219     if(maxThreads % T_B > 0)
220         B++;
221     ConvGPUshared<<<B, T_B, L*sizeof(cufftComplex)>>>(dev_myConv2, dev_mySignal2,
222                 dev_myFilter2, N, L,maxThreads);
223
224     cudaMemcpy(myConv2, dev_myConv2, C*numBatches*sizeof(cufftComplex),
225                cudaMemcpyDeviceToHost);
226
227     /**
228      * Frequency-domain Convolution GPU
229      */
230     cudaMemcpy(dev_mySignal3, mySignal1_pad, M*numBatches*sizeof(cufftComplex),
231                cudaMemcpyHostToDevice);
232     cudaMemcpy(dev_myFilter3, myFilter1_pad, M*numBatches*sizeof(cufftComplex),
233                cudaMemcpyHostToDevice);
234
235     cufftExecC2C(plan, dev_mySignal3, dev_mySignal3, CUFFT_FORWARD);
236     cufftExecC2C(plan, dev_myFilter3, dev_myFilter3, CUFFT_FORWARD);
237
238     maxThreads = M*numBatches;
239     T_B = 96;
240     B = maxThreads/T_B;
241     if(maxThreads % T_B > 0)
242         B++;
243     PointToPointMultiply<<<B, T_B>>>(dev_mySignal3, dev_myFilter3, maxThreads);
244     cufftExecC2C(plan, dev_mySignal3, dev_mySignal3, CUFFT_INVERSE);
245
246     T_B = 640;
247     B = maxThreads/T_B;
248     if(maxThreads % T_B > 0)
249         B++;
250     float scalar = 1.0/((float)M);
251     ScalarMultiply<<<B, T_B>>>(dev_mySignal3, scalar, maxThreads);
252
253     cudaMemcpy(myConv3, dev_mySignal3, M*numBatches*sizeof(cufftComplex),
254                cudaMemcpyDeviceToHost);
255
256     cufftDestroy(plan);
257
258     // Free vectors on CPU
259     free(mySignal1);
260     free(myFilter1);
261     free(myConv1);
262     free(myConv2);
263     free(myConv3);
264
265     // 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

Each equalizer in the PAQ system presents an interesting challenge from a GPU implementation perspective. The equations for each equalizer in Section 2.4 were reformulated in preparation for fast and efficient GPU implementation. This chapter is explain how the FIR equalizer filter coefficients are computed and applied.

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

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

Note that the “numerically optimized” detection filter  $h_{NO}$  and the SOQPSK-TG power spectral density  $\Psi$  are defined constants. The SOQPSK-TG power spectral density  $\Psi$  and  $H_{NO}$  are pre-computed and stored where  $H_{NO}$  is the 16,384 point FFT of  $h_{NO}$ . Applying  $h_{NO}$  in the frequency domain does not require an extra FFT, only extra complex multiplies.

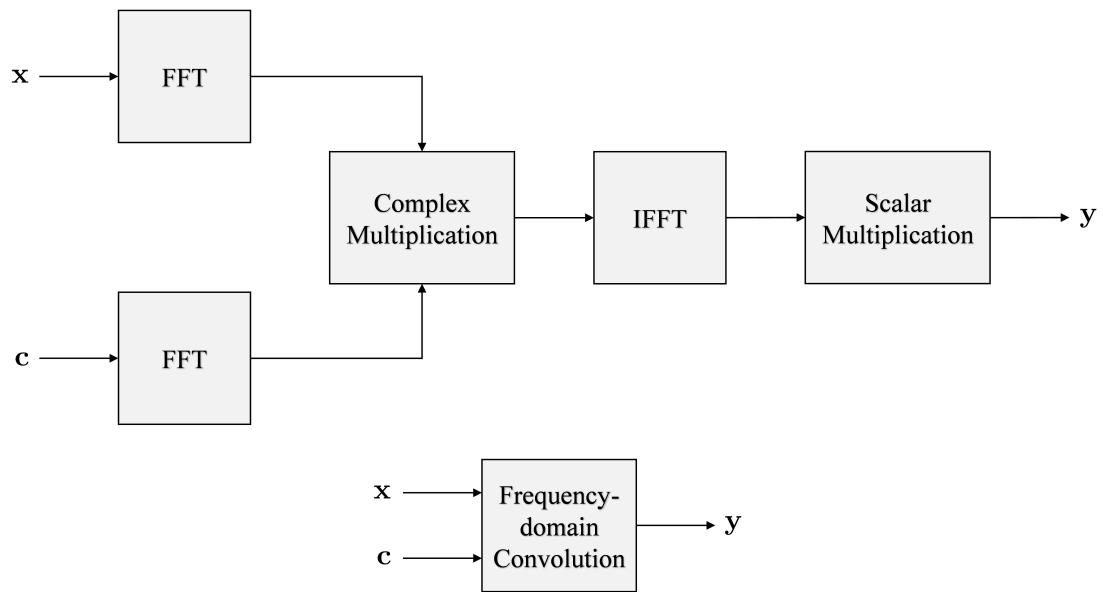


Figure 4.1: To simplify block diagrams, frequency-domain convolution is shown as one block.

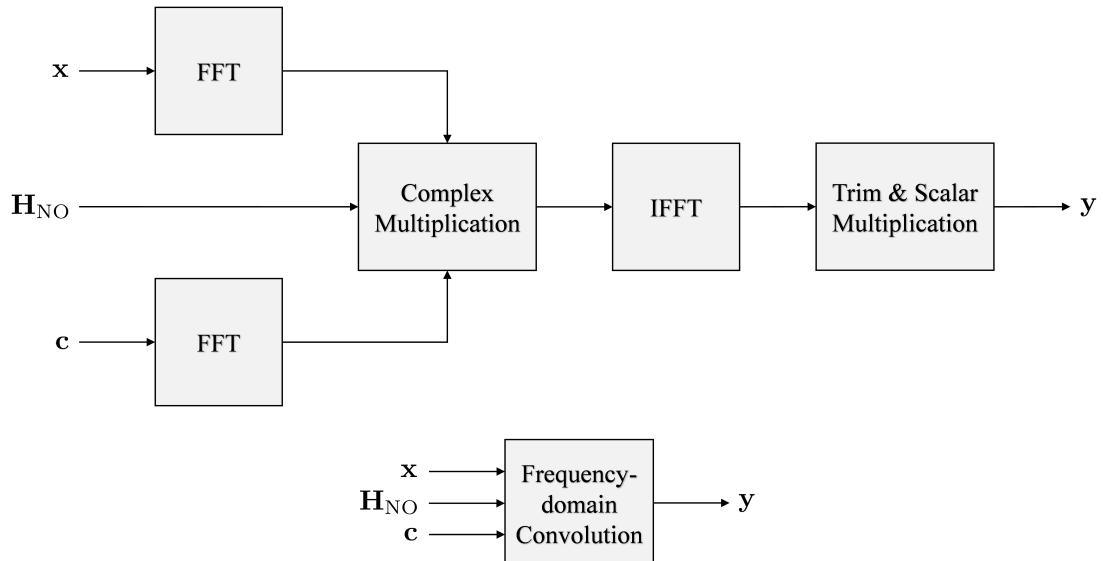


Figure 4.2: To simplify block diagrams, frequency-domain cascaded convolution is shown as one block.

## 4.1 Zero-Forcing and MMSE GPU Implementation

The ZF and MMSE FIR equalizer filter coefficient computations have exactly the same form as shown in Equations (2.18) and (2.26). Any ZF equalizer filter computation restructuring can also be applied directly to the MMSE equalizer filter computation. To simplify the conversation, the following section will explain the investigation only the ZF equalizer filter GPU implementation.

The ZF equalizer filter can be computed two ways: solve the system of linear equations

$$\mathbf{R}_{\hat{h}} \mathbf{c}_{\text{ZF}} = \hat{\mathbf{h}}_{n_0} \quad (4.1)$$

or compute the inverse of  $\mathbf{R}_{\hat{h}}$  then perform matrix vector multiplication

$$\mathbf{c}_{\text{ZF}} = \mathbf{R}_{\hat{h}}^{-1} \hat{\mathbf{h}}_{n_0}. \quad (4.2)$$

Both techniques require  $\mathcal{O}(n^3)$  operations making the computation of the ZF and MMSE equalizer filter coefficients extremely heavy. Computing a matrix inverse or solving linear systems in GPUs is especially challenging because common algorithms for matrix inversion and solving linear systems are serial. Three approaches to computing the equalizer filter coefficients were explored

- Using the Levinson-Durbin recursion algorithm to solve the system of equations
- Using the cuBLAS LU decomposition library to compute the inverse and matrix vector multiplication
- Using the cuSolver library to solve the system of equations.

The Levinson-Durbin recursion algorithm avoids  $\mathcal{O}(n^3)$  inverse or solving operations by leveraging the Toeplitz or diagonal-constant structure of  $\mathbf{R}_{\hat{h}}$  [19, Chap. 5]. The first GPU implementation of the Levinson-Durbin recursion algorithm computed the ZF equalizer filter assuming the matrix  $\mathbf{R}_{\hat{h}}$  and the vector  $\hat{\mathbf{h}}_{n_0}$  were *real-valued*. The Levinson-Durbin recursion algorithm showed promise by computing 3104 real-valued ZF equalizer filters in 500 ms.

The GPU implementation of Levinson-Durbin recursion was then converted from computing real-valued ZF equalizer filters to computing *complex-valued* ZF equalizer filters. Converting from real-valued to complex-valued filters requires more memory accesses and flops. The Levinson-Durbin recursion computed 3104 complex-valued ZF equalizer filters in 2,500 ms, in excess of the 1907 ms maximum for all processing time.

The next algorithm explored computed the inverse of  $\mathbf{R}_{\hat{h}}$  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 the linear system in Equation (4.1) using a GPU sparse complex-valued solver library called cuSolverSp. A sparse solver can be used because section 2.4.1 showed that  $\mathbf{R}_{\hat{h}}$  is sparse comprising 63% zeros because the matrix comprises of the vector  $\mathbf{r}_{\hat{h}}$  with finite support on  $-(L_{ch} - 1) \leq k \leq L_{ch} - 1$ .

“cusolverSpCcsqrsvBatched” is the GPU function used from the cuSolverSp library. cuSolverSpCcsqrsvBatched is a complex-valued batch solver that leverages the sparse properties of  $\mathbf{R}_{\hat{h}}$  by exploiting Compressed Row Storage (CRS) [20]. Compressed Row Storage reduces the large  $186 \times 186$  matrices to a 12544 element CSR matrix  $\mathbf{R}_{\hat{h}\text{CRS}}$ . Before cusolverSpCcsqrsvBatched can be called, the CSR matrix  $\mathbf{R}_{\hat{h}\text{CRS}}$  has to be built using  $\mathbf{r}_{\hat{h}}$ . An example of how to use the CUDA cusolverSp library can be found in [21]. Before the cuSolverSp library can be called, the CSR matrix  $\mathbf{R}_{\hat{h}\text{CRS}}$  must be built from the estimated channel auto-correlation and the vector  $\hat{\mathbf{h}}_{n_0}$  is built with the conjugated and time-reversed estimated channel  $\hat{\mathbf{h}}$ .

Figures 4.3 and 4.4 show how the ZF and MMSE equalizer filters are computed and applied to the received samples. 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.

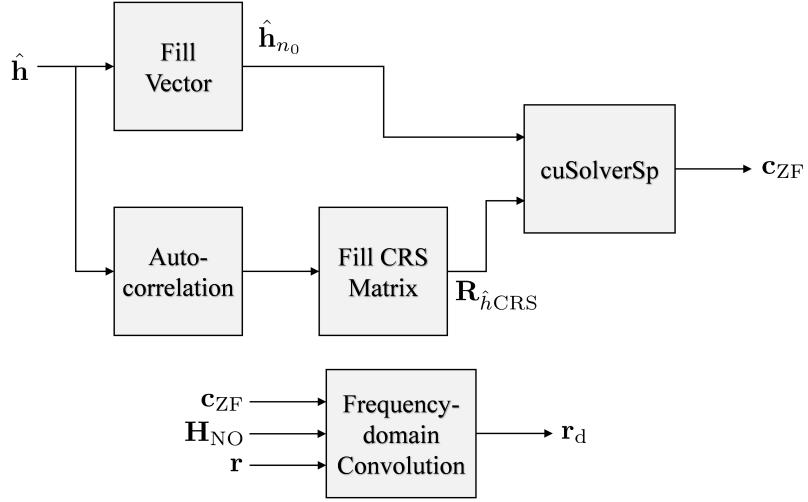


Figure 4.3: Block Diagram showing how the Zero-Forcing equalizer coefficients are implemented in the GPU.

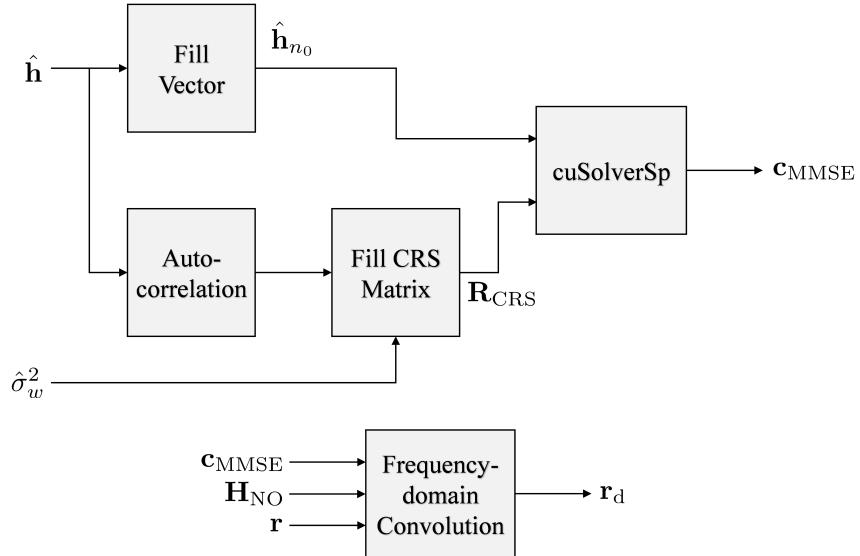


Figure 4.4: Block Diagram showing how the Minimum Mean Squared Error equalizer coefficients are implemented in the GPU.

Table 4.1: Three different algorithms were explored to compute the ZF and MMSE equalizer filters.

Algorithm	Data type	Execution Time (ms)
Levinson Recursion	floats	500
Levinson Recursion	Complex	2500
LU Decomposition	Complex	600
cuSolver	Complex	355.96

## 4.2 Constant Modulus Algorithm GPU Implementation

The Constant Modulus Algorithm (CMA) computes FIR equalizer filter coefficients by a steepest decent algorithm in Equation (2.33)

$$\mathbf{c}_{\text{CMA}}^{b+1} = \mathbf{c}_{\text{CMA}}^b - \mu \nabla J. \quad (4.3)$$

The more iterations a steepest decent algorithm executes, the better the CMA equalizer file will be. The cost function gradient used in the steepest decent algorithm is  $\nabla J$  shown in Equation (2.39). The most computationally heavy portion of the CMA equalizer filter implementation is computing the cost function gradient  $\nabla J$ . Section 2.4.2 showed there are two approaches to computing  $\nabla J$ : directly or using convolution.

To compute the cost function gradient directly the  $L_{pkt}$  sample summation

$$\nabla J = \frac{1}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} z(n) \mathbf{r}^*(n) \quad (4.4)$$

computes the cost function gradient where

$$z(n) = 2 \left[ y(n) y^*(n) - 1 \right] y(n). \quad (4.5)$$

This approach did not allow for multiple iterations because each equalizer filter coefficient required a 12672-sample summation. The summation in GPU kernels performs poorly because every equalizer filter coefficient accesses the full packet of received samples. One CMA iteration took 421.317 ms to execute computing  $\nabla J$  directly also applying  $\mathbf{c}_{\text{CMA}}^b$  and computing  $\mathbf{c}_{\text{CMA}}^{b+1}$ .

Using convolution to compute  $\nabla J$  decreased execution time significantly for the CMA equalizer filter. The cost function gradient was computed using

$$\nabla J(k) = \frac{1}{L_{pkt}} m(k), \quad -L_1 \leq k \leq L_2 \quad (4.6)$$

Table 4.2: Two different algorithms were explored to compute the cost function gradient  $\nabla J$ .

CMA Iteration Algorithm	Execution Time (ms)
$\nabla J$ directly	421.317
$\nabla J$ using convolution	88.774

where

$$m(n) = \sum_{m=0}^{L_{pkt}-1} z(m)\rho(n-m) \quad (4.7)$$

and

$$\rho(n) = r^*(n). \quad (4.8)$$

One CMA iteration took 88.774 ms to execute computing  $\nabla J$  using frequency-domain convolution also applying  $\mathbf{c}_{\text{CMA}}^b$  and computing  $\mathbf{c}_{\text{CMA}}^{b+1}$ . Note that all other frequency-domain convolution in this thesis  $2^{14}$  or 16,384 points, but the convolution length required to compute  $\nabla J(k)$  is  $12672 + 12672 - 1$  which is greater than 16,384. The FFTs in the computation of  $\nabla J(k)$  are  $2^{15}$  or 32,768 point FFTs.

Figure 4.5 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 4.2 lists the comparison on computing  $\nabla J(k)$  verse using convolution. By reformulating the computation of  $\nabla J$ , the execution time was reduced by a factor of 4.74. Implementing  $\nabla J$  directly only provided time for 2 iterations while using convolution to compute  $\nabla J(k)$  provided time for 12 iterations.

### 4.3 Frequency Domain Equalizer One and Two GPU Implementation

The Frequency Domain Equalizers (FDEs) were by far the fastest and easiest to implement in GPUs. The FDE1 and FDE2 block diagrams look just like the frequency-domain convolution

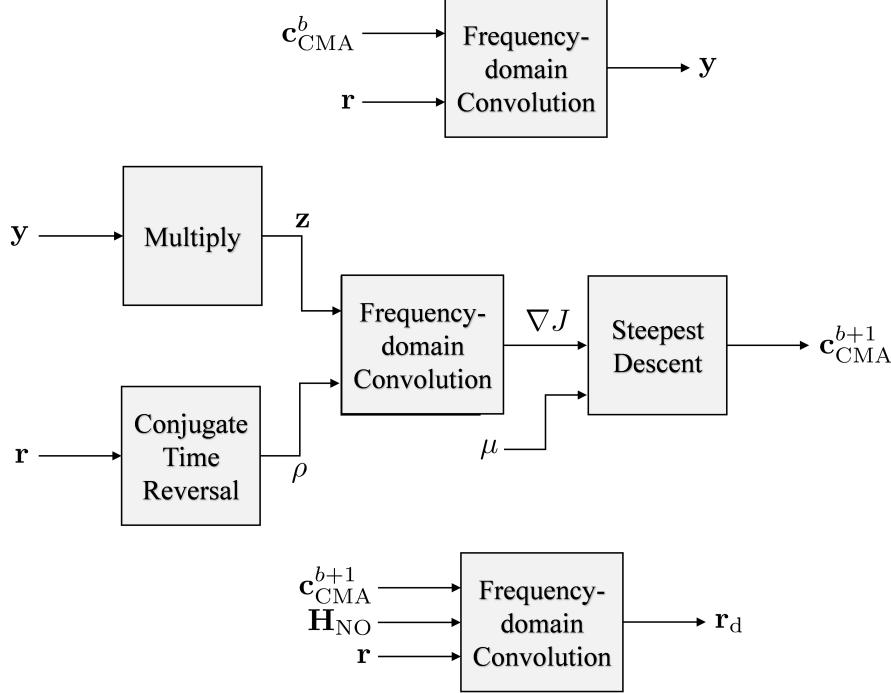


Figure 4.5: Block Diagram showing how the CMA equalizer filter is implemented in the GPU using frequency-domain convolution twice per iteration.

block diagram in Figure 4.2 except that complex multiplication is now complex multiplication and division.

### 4.3.1 Frequency Domain Equalizer One

FDE1 is the frequency-domain equivalent to the MMSE equalizer from Equation (2.46)

$$C_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{\text{FFT}}} \text{ for } k = 0, 1, \dots, N_{\text{FFT}} - 1 \quad (4.9)$$

where  $N_{\text{FFT}} = 16,384$  points. Since FDE1 is applied in the frequency domain, the FFT of the received signal  $R(e^{j\omega_k})$  is computed and the signal is equalized by performing a  $N_{\text{FFT}}$  point-to-point complex multiplication and division. The pre-computed FFT of the detection filter  $\hat{H}^*(e^{j\omega_k})$  is also applied in the point-to-point complex multiplication and division. The FFT of the equalized

Table 4.3: 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

and detected signal is

$$R_{d1}(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})H_{NO}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{FFT}} \text{ for } k = 0, 1, \dots, N_{FFT} - 1. \quad (4.10)$$

### 4.3.2 Frequency Domain Equalizer Two

FDE2 is the frequency-domain equivalent to the MMSE equalizer from Equation (2.47)

$$C_{FDE2}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{FFT}} \text{ for } k = 0, 1, \dots, N_{FFT} - 1 \quad (4.11)$$

where  $N_{FFT} = 16,384$  points. Since FDE2 is applied in the frequency domain, the FFT of the received signal  $R(e^{j\omega_k})$  is computed and the signal is equalized by performing a  $N_{FFT}$  point-to-point complex multiplication and division. The pre-computed FFT of the detection filter  $\hat{H}^*(e^{j\omega_k})$  is also applied in the point-to-point complex multiplication and division. The FFT of the equalized and detected signal is

$$R_{d1}(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})H_{NO}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \omega_k = \frac{2\pi}{N_{FFT}} \text{ for } k = 0, 1, \dots, N_{FFT} - 1. \quad (4.12)$$

Figures 4.6 and 4.7 show the block diagrams for GPU implementation of FDE1 and FDE2. As expected, these figures look just like the frequency-domain convolution block diagrams shown in Figure 4.2. Table 4.3 shows the execution times for calculating and applying FDE1 and FDE2.

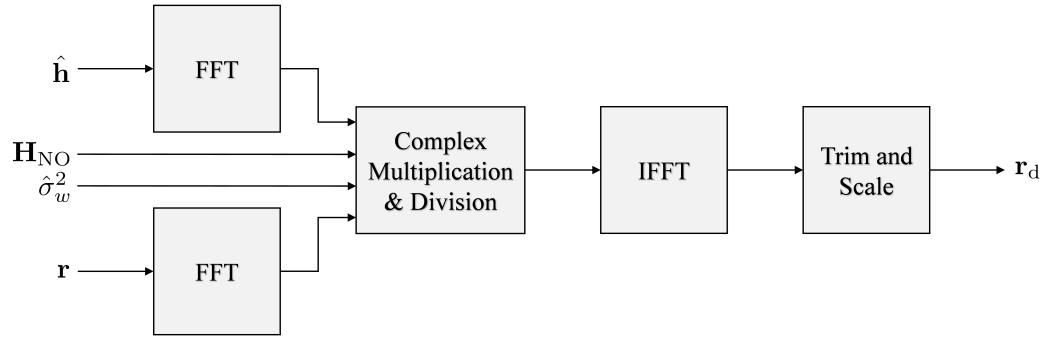


Figure 4.6: Block diagram showing Frequency Domain Equalizer One is implemented in the frequency domain in GPUs.

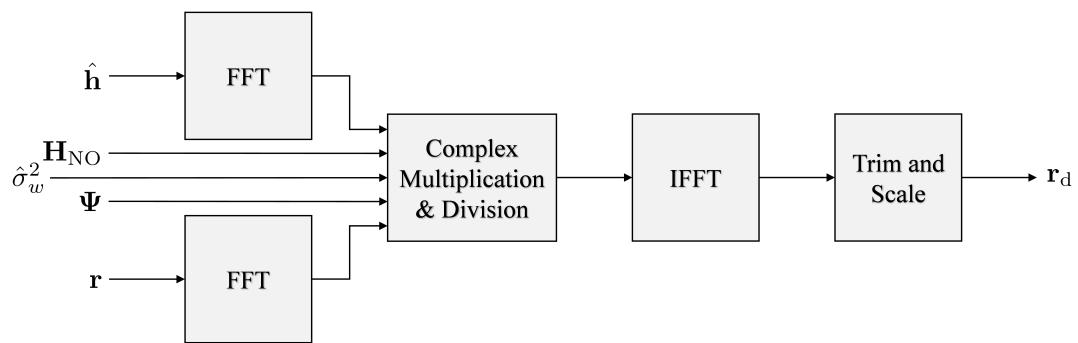


Figure 4.7: Block diagram showing Frequency Domain Equalizer Two is implemented in the frequency domain in GPUs.

## **Chapter 5**

### **Summary and Conclusions**

#### **5.1 GPU Implementation**

GPU Implementation Based on measured execution times of GPU kernels, multiple data-aided equalization filters where 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 time.

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. As of this writing the FDE1 equalizer is recommended, marking the best tradeoff between performance and computational complexity.

#### **5.2 Contributions**

Through the years of working on the PAQ project I:

1. Implemented algorithms using batch GPU libraries to reduce execution time on average by 5.

2. Reduced GPU convolution time by using batch GPU librarys and cascading filters in the frequency domain.
3. Implemented linear solver libraries to make ZF and MMSE equalizers feasible and real-time.
4. Reformulated the CMA to leverage the speed of GPU convolution.
5. Implemented new ADC to drive success of the PAQ project.
6. Implemented resampling polyphase filters in GPUs.
7. Participated heavily in flight tests at Edwards AFB.
8. Presented a paper frequency offset compensation for equalized SOQPSK at International Telemetering Conference (ITC) [22].

### 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  $\hat{\mathbf{h}}_{n_0}$ .

## Bibliography

- [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, 13
- [3] M. Rice, “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
- [4] ——, *Digital Communications: A Discrete-Time Approach*. Upper Saddle River, NJ: Pearson Prentice-Hall, 2009. 6
- [5] ——, “Phase 1 report: Preamble assisted equalization for aeronautical telemetry (PAQ), Brigham Young University,” Technical Report, 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 online at: <http://hdl.lib.byu.edu/1877/3242>, Tech. Rep., 2014. 9, 16, 17, 19
- [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] E. Perrins, “FEC systems for aeronautical telemetry,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 49, no. 4, pp. 2340–2352, October 2013. 13
- [9] 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. 23, 24
- [10] 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, 2006. 23

- [11] H. Sari, G. Karam, and I. Jeanclaud, “Frequency-domain equalization of mobile radio and terrestrial broadcast channels,” in *Global Telecommunications Conference, 1994. GLOBE-COM'94. Communications: The Global Bridge.*, IEEE. IEEE, 1994, pp. 1–5. 23
- [12] B. Ng, C.-T. Lam, and D. Falconer, “Turbo frequency domain equalization for single-carrier broadband wireless systems,” *IEEE transactions on wireless communications*, vol. 6, no. 2, 2007. 23
- [13] N. Al-Dhahir, M. Uysal, and H. Mheidat, “Single-carrier frequency domain equalization,” 2008. 23
- [14] J. Proakis and M. Salehi, *Digital Communications*, 5th ed. New York: McGraw-Hill, 2008. 23
- [15] Wikipedia, “Graphics processing unit,” 2015. [Online]. Available: [http://en.wikipedia.org/wiki/Graphics\\_processing\\_unit](http://en.wikipedia.org/wiki/Graphics_processing_unit) 27
- [16] ——, “Fastest fourier transform in the west,” 2017. [Online]. Available: <http://www.fftw.org/> 44
- [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. 44
- [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. 51
- [19] M. Hayes, *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996. 73
- [20] Wikipedia, “Sparse matrix,” 2017. [Online]. Available: [https://en.wikipedia.org/wiki/Sparse\\_matrix](https://en.wikipedia.org/wiki/Sparse_matrix) 74
- [21] NVIDIA, “Cuda toolkit documentation,” 2017. [Online]. Available: <http://docs.nvidia.com/cuda/> 74
- [22] J. Ravert and M. Rice, “On frequency offset compensation for equalized SOQPSK,” in *Proceedings of the International Telemetering Conference*, Glendale, AZ, October 2016. 82