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

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. An 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. Counter intuitive to the simulation results, the flight tests at Edwards AFB in March and June showed blind equalization is superior to data-aided equalization. Lab tests revealed some types of multipath caused timing loops in the RF receiver to produce garbage samples. Data-aided equalizers based on data-aided channel estimation leads to high bit error rates. A new experimental setup is been proposed, replacing the RF receiver with a RF data acquisition card. The data acquisition card will always provide good samples because the card has no timing loops, regardless of severe multipath.

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

List of Tables **ix**

List of Figures **xi**

1 Introduction **1**

1.1 Multipath in Aeronautical Telemetry 1

1.2 Preamble Assisted Equalization Project 3

2 PAQ Equations **9**

2.1 Estimators 9

2.1.1 Preamble Detection 9

2.1.2 Frequency Offset Compensation 11

2.1.3 Channel Estimation 12

2.1.4 Noise Variance Estimation 12

2.1.5 Symbol-by-Symbol Detector 13

2.2 Equalizers 14

2.2.1 Zero-Forcing and Minimum Mean Square Error Equalizers 14

2.2.2 The Constant Modulus Algorithm 19

2.2.3 The Frequency Domain Equalizers 23

3 Signal Processing in GPUs **25**

3.1 GPU and CUDA Introduction 26
3.1.1 CPU verse CUDA example 26
3.1.2 GPU kernel using threads and thread blocks 29
3.1.3 GPU Memory 30
3.1.4 Thread Optimization 32
3.1.5 CPU and GPU Pipelining 35
3.2 GPU Convolution 40
3.2.1 Floating Point Operation Comparison 41
3.2.2 CPU and GPU Single Batch Convolution Comparison 42
3.2.3 Batched Convolution 48

4 Equalizer GPU Implementation **67**

4.1 Zero-Forcing and MMSE GPU Implementation 69
4.2 Constant Modulus Algorithm GPU Implementation 72
4.3 Frequency Domain Equalizer One and Two GPU Implementation 74

5 Final Summary **77**

Bibliography **78**

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. 33

3.2 Defining start and stop lines for timing comparison in Listing 3.5. 44

3.3 Convolution computation times with signal length 12672 and filter length 186 on a Tesla K40c GPU. 46

3.4 Convolution computation times with signal length 12672 and filter length 21 on a Tesla K40c GPU. 47

3.5 Defining start and stop lines for execution time comparison in Listing 3.6. 49

3.6 Batched convolution execution times with for a 12672 sample signal and 186 tap filter on a Tesla K40c GPU. 51

3.7 Batched convolution execution times with for a 12672 sample signal and 21 tap filter on a Tesla K40c GPU. 51

3.8 Batched convolution execution times with for a 12672 sample signal and cascaded 21 and 186 tap filter on a Tesla K40c GPU. 53

4.1 Defining start and stop lines for timing comparison in Listing 3.5. 70

4.2 The gradient vector ∇J can be computed directly or using convolution. 73

4.3 Execution times for calculating and applying Frequency Domain Equalizer One and Two. 74

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

1.2 A block diagram of the physical PAQ project 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

1.3 A picture of the physical PAQ project hardware. Right: Components in the dashed and dotted box from Figure 1.2. Left: Components in dashed box in Figure 1.2. Note that the T/M Receiver is not pictured. 4

1.4 A diagram showing the PAQ packetized sample structure. 5

1.5 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 $r(n)$. 6

1.6 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. 6

1.7 A block diagram of the estimators in the PAQ project. 7

1.8 A block diagram of application of the FIR equalizer and detection filters in the Preamble Assisted Equalization (PAQ) project. 8

2.1 Offset Quadriture Phase Shift Keying symbol by symbol detector. 13

2.2 Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$. 22

2.3 SOQPSK-TG power spectral density. 24

3.1 NVIDIA Tesla K40c and K20c. 26

3.2 A block diagram of how a CPU sequentially performs vector addition. 27

3.3 A block diagram of how a GPU performs vector addition in parallel. 27

3.4 32 threads launched in 4 thread blocks with 8 threads per block. 30

3.5 36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads. 30

3.6 Diagram comparing memory size and speed. Global memory is massive but extremely slow. Registers are extremely fast but there are very few. 31

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. 31

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. 32

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. 35

3.10 Plot showing the number of threads per block doesn't always drastically affect execution time. 36

3.11 The typical approach of CPU and GPU operations. This block diagram shows the profile of Listing 3.3. 36

3.12 GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 3.4. 37

3.13 A block diagram of pipelining a CPU with three GPUs. 38

3.14 Block diagrams showing showing time-domain convolution and frequency-domain convolution. 41

3.15 Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 186 tap complex filter. 43

3.16 Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 21 tap complex filter. 44

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. 45

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. 46

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. 47

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. 48

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. 49

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. . . . 50

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. 51

3.24 Comparison of a batched complex convolution on a GPU. The signal length is variable and the filter is fixed at 186 taps. 52

3.25 Comparison of a batched complex convolution on a GPU. The signal length is variable and the filter is fixed at 21 taps. 53

3.26 Comparison of a batched complex convolution on a GPU. The filter length is variable and the signal length is set to 12672 samples. 54

3.27 Block diagrams showing showing cascaded time-domain convolution and frequency-domain convolution. 55

4.1 To simplify block diagrams, frequency-domain convolution is shown as one block. 68

4.2 To simplify block diagrams, frequency-domain cascaded convolution is shown as one block. 68

4.3 Block Diagram showing how the Zero-Forcing equalizer coefficients are implemented in the GPU. 71

4.4 Block Diagram showing how the Minimum Mean Squared Error equalizer coefficients are implemented in the GPU. 71

4.5 Block Diagram showing how the CMA equalizer filter is implemented in the GPU using frequency-domain convolution twice per iteration. 73

4.6 Diagram showing Frequency Domain Equalizer One is implemented in the frequency domain in GPUs. 75

4.7 Diagram showing Frequency Domain Equalizer Two is implemented in the frequency domain in GPUs. 75

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. Receivers use equalizers to combat multipath but often the transmitted information can not be recovered. 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 impulse response filters (FIR). The impulse response of the FIR filters are computed based on the estimated parameters to better mitigate multipath.

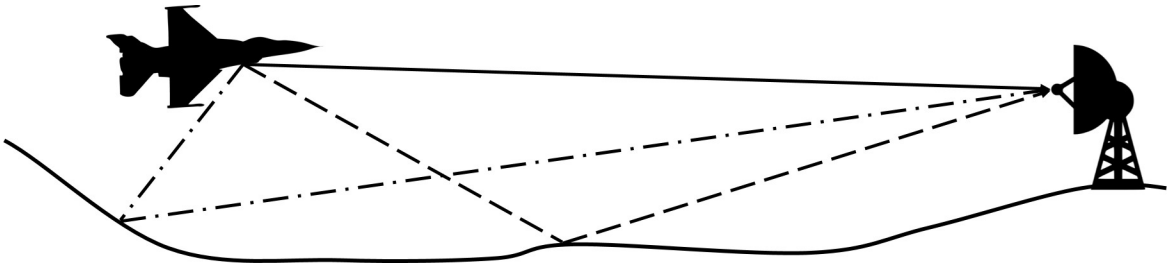


Figure 1.1: Multipath can occur when a signal is received multiple paths like line-of-sight or ground bounce or reflections.

1.2 Preamble Assisted Equalization Project

Data-aided equalization in aeronautical telemetry has been studied and tested by the Preamble Assisted Equalization (PAQ) project [3]. The PAQ project built a TRL 6 system that compares five data-aided equalizers to blind equalization and no equalization [4]. Bit error statistics were be used as the figure or merit for the equalization algorithms. Live flight tests were conducted at Edwards AFB in March and June 2016. The five data-aided equalizers the PAQ project 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).

Hardware

A block diagram of the PAQ project physical system is shown in Figure 1.2. A picture of the physical components is shown in Figure 1.3. The major components, and their functions are summarized in the following.

- The **T/M mixer** down-converts from L or C band RF to IF (70 MHz) 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.
- The **ADC** produces 14-bit samples of the real-valued bandpass signal centered at IF sampled at $93^1/_3$ 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.

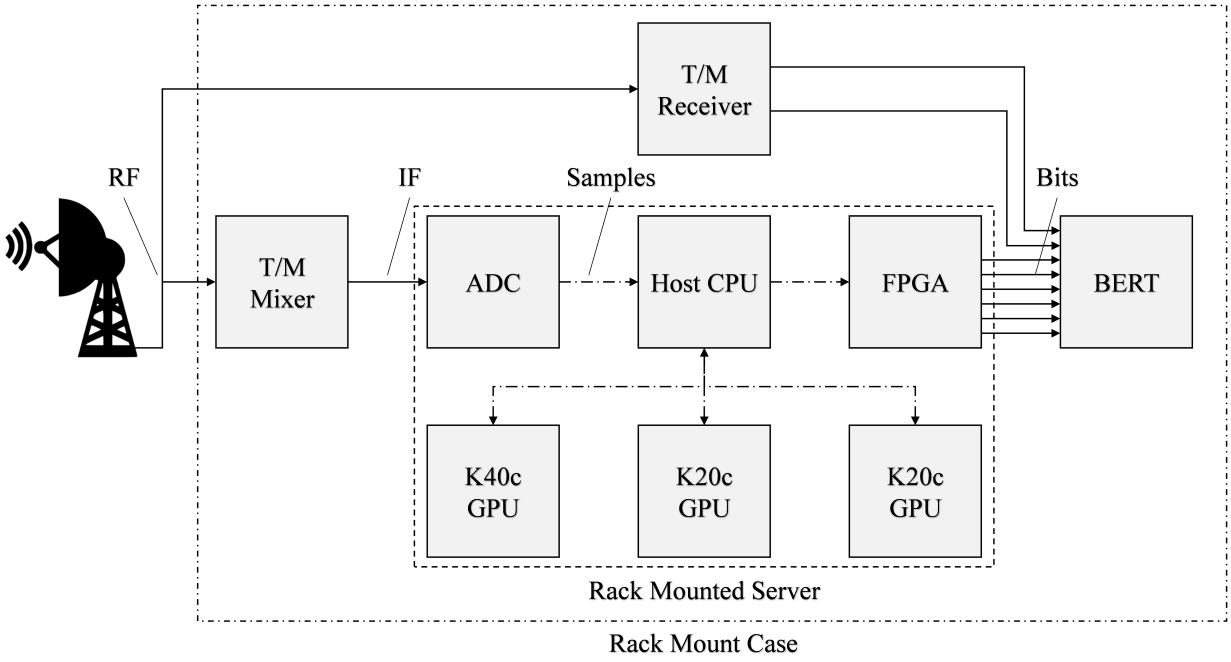


Figure 1.2: A block diagram of the physical PAQ project hardware. The components inside the rack mounted server are in the dashed box. All the components in the dashed and dotted box are housed in a rack mounted case.

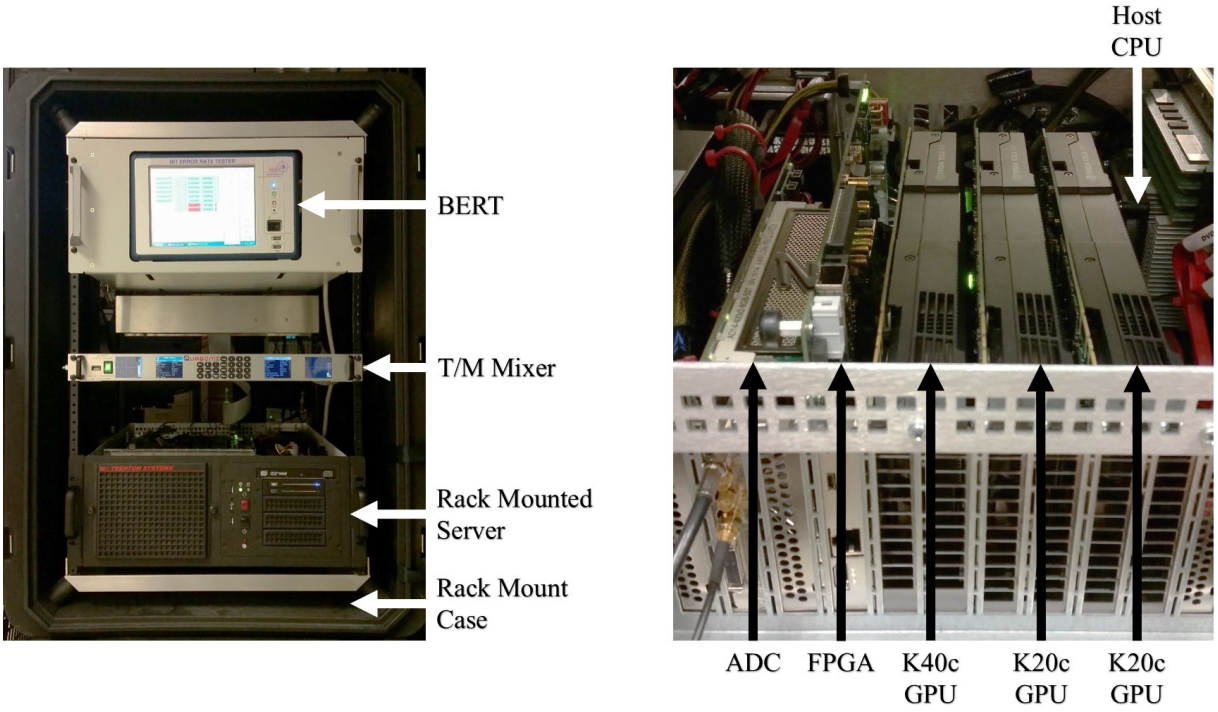


Figure 1.3: A picture of the physical PAQ project hardware. Right: Components in the dashed and dotted box from Figure 1.2. Left: Components in dashed box in Figure 1.2. Note that the T/M Receiver is not pictured.

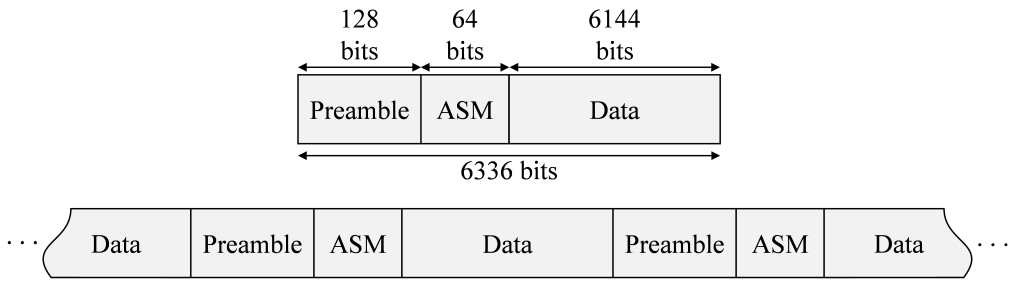


Figure 1.4: A diagram showing the PAQ packetized sample structure.

- The three **GPUs** are where all the detection, estimation, equalization and demodulation resides. While the CPU has one to eight powerful processors, GPUs have thousands of small less powerful processors that work in parallel. The signal processing is done in GPUs rather than FPGAs or a CPU because programming GPUs is faster and easier than programming FPGAs and CPUs do not prosess the required processing power.

- The **FPGA** receives all the bit streams from the host CPU via the PCIe bus then clocks each stream out in parallel to the BERT for BER testing.

- The bit error rate tester (**BERT**) counts the errors in each input bit stream by comparing the streams to a PN sequence.

- The **T/M Receiver** outputs bit streams for blind equalization and no equalization for BER comparison.

To enable data-aided equalization, the PAQ project bit stream has a packetized structure shown in Figure 1.4. 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

$$034776\text{C}7272895\text{B}0_{\text{hex}}. \tag{1.1}$$

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 modulate

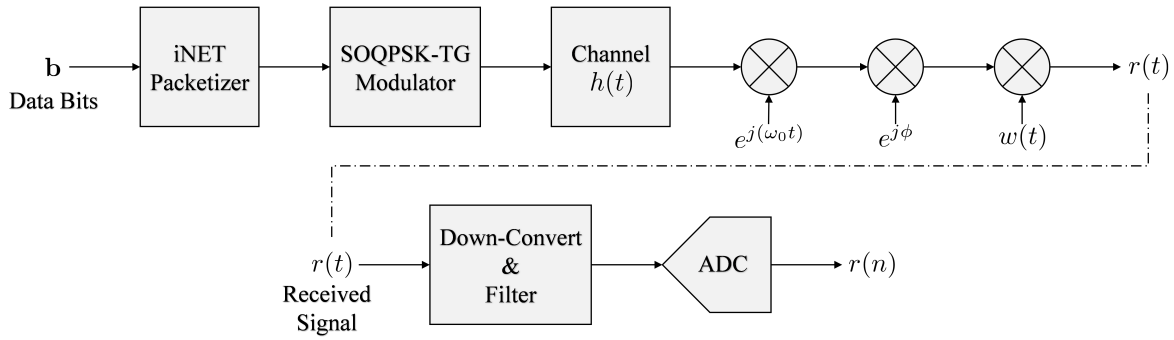


Figure 1.5: Received signal has multipath interference, frequency offset, phase offset and additive white Gaussian noise. The received signal is down-converted filtered and sampled to produce the sample sequence $r(n)$.

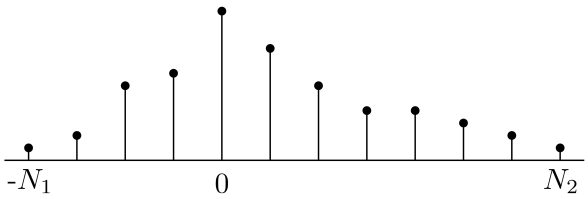


Figure 1.6: An illustration of the discrete-time channel of length $N_1 + N_2 + 1$ with a non-causal component comprising N_1 samples and a causal component comprising N_2 samples.

a SOQPSK-TG carrier at 10 Mbits/second. With the preamble and ASM periodically inserted, the over the air bit rate is 10.3125 Mbits/second.

After modulation, the transmitted signal experiences multipath interference modeled as the channel $h(t)$. The transmitted signal also experiences a frequency offset ω_0, a phase offset ϕ and additive white Gaussian noise $w(t)$. The received signal is down-converted, filtered in the T/M mixer, sampled at $93\frac{1}{3}$ Msamples/second by the ADC then down-converted again to baseband and resampled by $\frac{99}{448}$ in the GPUs resulting in the sampled sequence $r(n)$ at rate 20.625 Msamples/second or 2 samples/bit.

The model of the received signal is shown in Figure 1.5. 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 1.6 shows the full discrete-time $L_h = N_1 + N_2 + 1$ sample channel. The received signal is sampled at 20.625 Msamples/second. The iNET packet is $L_{\text{pkt}} = 12672$ samples long with the preamble $L_{\text{p}} = 256$ samples and the ASM $L_{\text{ASM}} = 128$ samples.

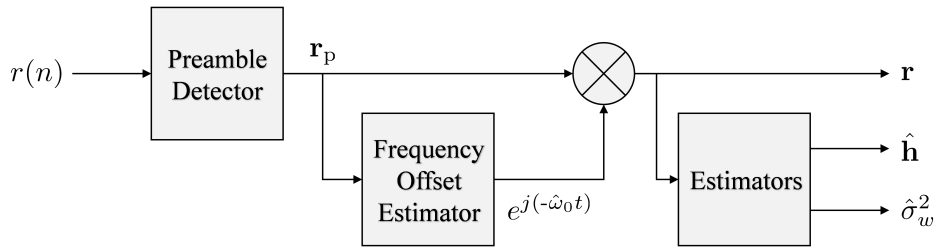


Figure 1.7: A block diagram of the estimators in the PAQ project.

Digital Signal Processing

A high-level digital signal processing flow is shown in Figure 1.7 and 1.8. 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 correlates received samples with L_{P} samples of a locally stored copy of the pilot in (1.2). The preamble detector block outputs the vector of samples $\mathbf{r}_{\mathrm{p}}$ with the iNET packetized structure. The first $L_{\mathrm{P}} + L_{\mathrm{ASM}}$ samples in $\mathbf{r}_{\mathrm{p}}$ correlate with the received pilot samples.

$$\mathbf{p} = \begin{bmatrix} p(0) & p(1) & \cdots & p(L_{\mathrm{P}} + L_{\mathrm{ASM}} - 1) \end{bmatrix} \tag{1.2}$$

The located preamble samples are used first to estimate the frequency offset. The estimated frequency offset $\hat{\omega}_0$ rads/sample is then used to "de-rotate" the vector of samples $\mathbf{r}_{\mathrm{p}}$ to produce $\mathbf{r}$. The de-rotated samples in the vector $\mathbf{r}$ that correlate to the preamble and ASM are used to estimate the channel $\hat{\mathbf{h}}$ and noise variance $\hat{\sigma}_w^2$. The channel and noise variance estimates are done in the estimators block.

Equipped with knowledge of the estimated channel and noise variance, data-aided finite impulse response (FIR) equalizer filters can be computed. The all the data-aided equalizer filters are computed using the same channel and noise variance estimates. The blocks shown in Figure 1.8 are duplicated in five independent branches producing five estimated vectors of bits $\hat{\mathbf{b}}$, one for each equalizer.

The PAQ project designed the data-aided FIR equalizer filters to be 5 times longer than the channel estimate. The equalizer filter has a non-causal component comprising $L_1 = 5N_1 = 60$

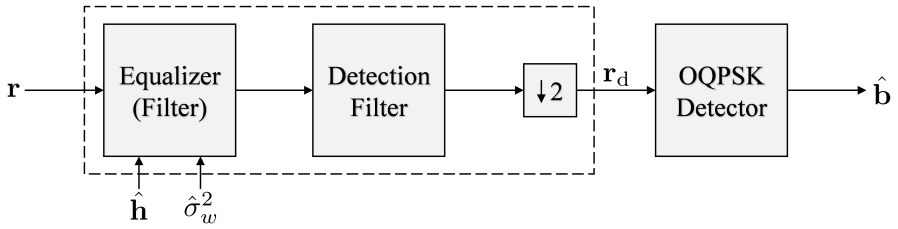


Figure 1.8: A block diagram of application of the FIR equalizer and detection filters in the Preamble Assisted Equalization (PAQ) project.

samples and a causal component comprising $L_2 = 5N_2 = 125$ samples. The $L_{\text{EQ}} = L_1 + L_2 + 1$ sample full discrete-time equalizer filters are computed and applied in the equalizer filter block.

The output of the equalizer filters are then filtered by a SOQPSK-TG detection filter and down-sampled by 2 in preparation for an OQPSK detector. The $\mathbf{r}_{\text{d}}$ in each equalizer branch has a sample rate of 1 sample/bit or 2 samples/symbol. The OQPSK detector block outputs the vector of estimated bits $\hat{\mathbf{b}}$. Finally the BER for each equalizer is obtained by comparing the vectors of estimated bits $\hat{\mathbf{b}}$ to the PN sequence.

The GPUs in Figure 1.2 and 1.3 perform all the digital signal processing in parallel. To introduce as much parallelism as possible, the received samples are processed in 39,321,600 sample sets. At 20.625 Msamples/second, each set of 39,321,600 samples is 1907 milliseconds worth of data. Each set has at most 3104 independent 12672 sample iNET packets. The GPU processes 3104 packets in parallel by leveraging batched processing. Each packet is a batch and each batch performs the algorithms shown in Figures 1.7 and 1.8. In order to stay real-time, **all** processing must be completed in 1907 ms.

This thesis, will illustrate how the five PAQ data-aided equalizers were computed and applied to the received samples in GPUs. The dashed box in Figure 1.8 emphasizes which processing blocks are focused on.

Chapter 2 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

PAQ Equations

This thesis studies GPU implementation of the PAQ project data-aided FIR equalizer filters. The impulse response of the FIR equalizer filters are computed using channel and noise variance estimates based on the iNET preamble and ASM in the received signal. All the estimates are data-aided and thus require finding the preamble and the ASM in the received signal. A preamble detector is employed to estimate the start of each iNET packet in the set. Figure 1.7 shows a block diagram of the estimators in the PAQ project. The data-aided FIR equalizer filters are then computed and the received samples are equalized. With the received samples equalized, a detection filter is applied and the symbols are estimated using a OQPSK detector. Figure 1.8 shows a block diagram of the FIR filters and symbol detector in the PAQ project.

The estimators, detection filter and OQPSK detector will be briefly explained in Section 2.1. Section 2.2 will explain the equations for the FIR equalizer filters and the application of the equalizers and detection filter.

2.1 Estimators

2.1.1 Preamble Detection

To compute data-aided equalizers, preambles in the received signal 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 $\mathbf{r}_{\text{p}}$. Each vector of samples $\mathbf{r}_{\text{p}}$ has the structure shown in Figure 1.4.

Before the structuring the received samples into packets, the preambles are found using the preamble detector explained in [5]. The preamble detector computes the function $L(n)$ for each sample in the set. Peaks in $L(n)$ identify the locations of a preamble or the start of a packet. The

function $L(n)$ is given by

$$L(n) = \sum_{m=0}^{7} \left[I^2(n,m) + Q^2(n,m) \right] \tag{2.1}$$

where

$$\begin{aligned} I(n,m) \approx & \sum_{\ell \in \mathcal{L}_1} r_R(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_2} r_R(\ell+32m+n) + \sum_{\ell \in \mathcal{L}_3} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_4} r_I(\ell+32m+n) \\ & + 0.7071 \left[\sum_{\ell \in \mathcal{L}_5} r_R(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_6} r_R(\ell+32m+n) \right. \\ & \left. + \sum_{\ell \in \mathcal{L}_7} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_8} r_I(\ell+32m+n) \right], \end{aligned} \tag{2.2}$$

and

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

with

$$\begin{aligned}
\mathcal{L}_1 &= \{0, 8, 16, 24\} \\
\mathcal{L}_2 &= \{4, 20\} \\
\mathcal{L}_3 &= \{2, 10, 14, 22\} \\
\mathcal{L}_4 &= \{6, 18, 26, 30\} \\
\mathcal{L}_5 &= \{1, 7, 9, 15, 17, 23, 25, 31\} \\
\mathcal{L}_6 &= \{3, 5, 11, 12, 13, 19, 21, 27, 28, 29\} \\
\mathcal{L}_7 &= \{1, 3, 9, 11, 12, 13, 15, 21, 23\} \\
\mathcal{L}_8 &= \{5, 7, 17, 19, 25, 27, 28, 29, 31\}.
\end{aligned} \tag{2.4}$$

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

$$\mathbf{r}_{\text{p}} = \begin{bmatrix} r(n) \\ \vdots \\ r(n + L_{\text{pkt}} - 1) \end{bmatrix} = \begin{bmatrix} r_{\text{p}}(0) \\ \vdots \\ r_{\text{p}}(L_{\text{pkt}} - 1) \end{bmatrix} \tag{2.5}$$

2.1.2 Frequency Offset Compensation

The frequency offset estimator shown in Figure 1.7 is the estimator taken from [6, eq. (24)]. With the notation adjusted slightly, the frequency offset estimate is

$$\hat{\omega}_0 = \frac{1}{L_q} \arg \left\{ \sum_{n=i+2L_q}^{i+7L_q-1} r_{\text{p}}(n) r_{\text{p}}^*(n - L_q) \right\} \quad \text{for } i = 1, 2, 3, 4, 5. \tag{2.6}$$

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

$$r(n) = r_{\text{p}}(n) e^{-j\hat{\omega}_0 n}. \tag{2.7}$$

Equations (2.6) and (2.7) are easily implemented into GPUs.

2.1.3 Channel Estimation

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

$$\hat{\mathbf{h}} = \underbrace{\left(\mathbf{X}^{\dagger}\mathbf{X}\right)^{-1}\mathbf{X}^{\dagger}}_{\mathbf{X}_{\text{lpi}}}\mathbf{r} \tag{2.8}$$

where

$$\mathbf{X} = \begin{bmatrix} p(N_2) & & & \\ \vdots & p(N_2) & & \\ p(L_{\text{p}} + L_{\text{ASM}} - N_1) & \vdots & \ddots & \\ & p(L_{\text{p}} + L_{\text{ASM}} - N_1) & p(N_2) & \\ & & \vdots & \\ & & p(L_{\text{p}} + L_{\text{ASM}} - N_1) & \end{bmatrix} \tag{2.9}$$

is a $(L_{\text{p}} + L_{\text{ASM}} - N_1 - N_2) \times (N_1 + N_2 + 1)$ convolution matrix formed from the ideal preamble and ASM samples. The $(N_1 + N_2 + 1) \times (L_{\text{p}} + L_{\text{ASM}} - N_1 - N_2)$ matrix $\mathbf{X}_{\text{lpi}}$ is the left pseudo-inverse of $\mathbf{X}$. The ML channel estimator is the result of the matrix operation

$$\hat{\mathbf{h}} = \mathbf{X}_{\text{lpi}}\mathbf{r}. \tag{2.10}$$

The matrix operation $\mathbf{X}_{\text{lpi}}\mathbf{r}$ is implemented simply and efficiently in GPUs.

2.1.4 Noise Variance Estimation

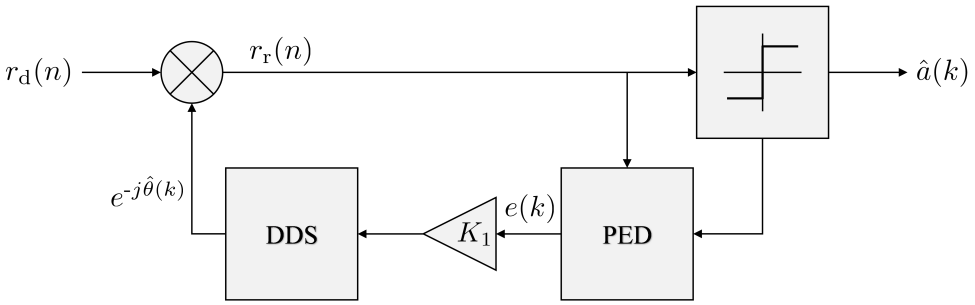
The noise variance estimator is also taken from [2, eq. 9]

$$\hat{\sigma}_w^2 = \frac{1}{2\rho}\left|\mathbf{r} - \mathbf{X}\hat{\mathbf{h}}\right|^2 \tag{2.11}$$

where

$$\rho = \text{Trace}\left\{\mathbf{I} - \mathbf{X}\left(\mathbf{X}^{\dagger}\mathbf{X}\right)^{-1}\mathbf{X}^{\dagger}\right\}. \tag{2.12}$$

Equation (2.11) is easily implemented into GPUs.



$$\hat{a}(k) = \begin{cases} p(k) & k < L_p + L_{\mathrm{asm}} \\ \mathrm{sgn}(\mathrm{Re}\{r_{\mathrm{r}}(k)\}) & k \geq L_p + L_{\mathrm{asm}} \quad \& \quad k \text{ even} \\ \mathrm{sgn}(\mathrm{Im}\{r_{\mathrm{r}}(k)\}) & k \geq L_p + L_{\mathrm{asm}} \quad \& \quad k \text{ odd} \end{cases}$$

$$e(k) = \begin{cases} 0 & k \text{ even} \\ \hat{a}(k-1)\mathrm{Im}\{r_{\mathrm{r}}(k-1)\} - \hat{a}(k)\mathrm{Re}\{r_{\mathrm{r}}(k)\} & k \text{ odd} \end{cases}$$

Figure 2.1: Offset Quadriture Phase Shift Keying symbol by symbol detector.

2.1.5 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 $L_{\mathrm{df}} = 21$ sample "numerically optimized" SOQPSK detection filter H_{NO} [7, Fig. 3]. The symbol-by-symbol detector block in Figure 1.8 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 efficency [7].

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. While phase offset, timing offset and multipath are combated with equalizers, batched based equalizers cannot remove residual frequency offset. 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 batches of data at a time by parallelizing on a packet by packet basis. 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.

2.2 Equalizers

This thesis examines that performance and GPU implementation of five FIR 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.

2.2.1 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

$$\mathbf{A}\mathbf{c} = \mathbf{b} \tag{2.13}$$

where $\mathbf{c}$ is a vector of desired equalizer coefficients and the 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

$$c_{\text{ZF}}(-L_1) \quad \cdots \quad c_{\text{ZF}}(0) \quad \cdots \quad c_{\text{ZF}}(L_2). \tag{2.14}$$

The filter coefficients are the solution to the matrix vector equation [8, eq. (311)]

$$\mathbf{c}_{\text{ZF}} = \left(\mathbf{H}^{\dagger}\mathbf{H}\right)^{-1}\mathbf{H}^{\dagger}\mathbf{u}_{n_0} \tag{2.15}$$

where

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

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \Bigg\} n_0 - 1 \text{ zeros} \\ \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \Bigg\} N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \tag{2.17}$$

where $n_0 = N_1 + L_1 + 1$ and

$$\mathbf{H} = \begin{bmatrix} \hat{h}(-N_1) & & & \\ \hat{h}(-N_1+1) & \hat{h}(-N_1) & & \\ \vdots & \vdots & \ddots & \\ \hat{h}(N_2) & \hat{h}(N_2-1) & & \hat{h}(-N_1) \\ & \hat{h}(N_2) & & \hat{h}(-N_1+1) \\ & & & \vdots \\ & & & \hat{h}(N_2) \end{bmatrix}. \tag{2.18}$$

Equation (2.15) can be implemented directly but there are many optimization that greatly reduce computation. The heaviest computation is the $\mathcal{O}(n^3)$ inverse operation followed by the $\mathcal{O}(n^2)$ matrix matrix multiplies. Rather than performing a heavy inverse, multiplying $\mathbf{H}^\dagger\mathbf{H}$ on both sides of equation (2.15) results in

$$\begin{aligned} \mathbf{H}^\dagger\mathbf{H}\mathbf{c}_{\text{ZF}} &= \mathbf{H}^\dagger\mathbf{u}_{n_0} \\ \mathbf{R}_{\hat{h}}\mathbf{c}_{\text{ZF}} &= \hat{\mathbf{h}}_{n_0} \end{aligned} \tag{2.19}$$

where

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

is the auto-correlation matrix of the channel estimate $\hat{\mathbf{h}}$ and

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

is a vector with the time reversed and conjugated channel estimate $\hat{\mathbf{h}}$ centered at n_0. The channel estimate auto-correlation sequence is

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

Note that the auto-correlation matrix $\mathbf{R}_{\hat{h}}$ is comprised of

$$\mathbf{r}_{\hat{h}} = \begin{bmatrix} r_{\hat{h}}(0) \\ \vdots \\ r_{\hat{h}}(L_{ch}) \\ r_{\hat{h}}(L_{ch}+1) \\ \vdots \\ r_{\hat{h}}(L_{eq}-1) \end{bmatrix} = \begin{bmatrix} r_{\hat{h}}(0) \\ \vdots \\ r_{\hat{h}}(L_{ch}) \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{2.23}$$

Using $\mathbf{r}_{\hat{h}}$ eliminates the need for matrix matrix multiply of $\mathbf{H}^\dagger\mathbf{H}$. Also, $r_{\hat{h}}(k)$ only has support on $-(L_{ch}-1) \leq k \leq L_{ch}-1$ making $\mathbf{R}_{\hat{h}}$ sparse or $\%63$ zeros. NEEDS CHECKING!!!! The sparseness of $\mathbf{R}_{\hat{h}}$ can be leveraged to reduce computation drastically.

MMSE Equalizer

The MMSE equalizer is an FIR filter defined by the coefficients

$$c_{\text{MMSE}}(-L_1) \quad \cdots \quad c_{\text{MMSE}}(0) \quad \cdots \quad c_{\text{MMSE}}(L_2). \tag{2.24}$$

The filter coefficients are the solution to the matrix vector equation [8, eq. (330) and (333)]

$$\mathbf{c}_{\text{MMSE}} = \left[\mathbf{G}\mathbf{G}^\dagger + 2\hat{\sigma}_w^2\mathbf{I}_{L_1+L_2+1}\right]^{-1}\mathbf{g}^\dagger \tag{2.25}$$

where $\mathbf{I}_{L_1+L_2+1}$ is the $(L_1+L_2+1)\times(L_1+L_2+1)$ identity matrix, $\hat{\sigma}_w^2$ is the estimated noise variance, $\mathbf{G}$ is the $(L_1+L_2+1)\times(N_1+N_2+L_1+L_2+1)$ matrix given by

$$\mathbf{G} = \begin{bmatrix} \hat{h}(N_2) & \cdots & \hat{h}(-N_1) & & & \\ & \hat{h}(N_2) & \cdots & \hat{h}(-N_1) & & \\ & & \ddots & & \ddots & \\ & & & \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \end{bmatrix} \tag{2.26}$$

and $\mathbf{g}^\dagger$ is the $(L_1+L_2+1)\times 1$ vector given by

$$\mathbf{g}^\dagger = \hat{\mathbf{h}}_{n0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix}. \tag{2.27}$$

Computing $\mathbf{c}_{\text{MMSE}}$ can be simplified by noticing that $\mathbf{g}^\dagger = \hat{\mathbf{h}}_{n_0}$, $\mathbf{G}\mathbf{G}^\dagger = \mathbf{R}_{\hat{h}}$ in Equation (2.20). To further simplify MMSE, twice the estimated noise variance is added down the diagonal of the channel estimate auto-correlation matrix

$$\mathbf{R} = \mathbf{R}_{\hat{h}} + 2\hat{\sigma_w^2}\mathbf{I}_{L_1+L_2+1} = \begin{bmatrix} r_h(0)+2\hat{\sigma_w^2} & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0)+2\hat{\sigma_w^2} & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0)+2\hat{\sigma_w^2} \end{bmatrix}. \tag{2.28}$$

By placing Equation (2.28) and (2.27) into (2.25) results in

$$\mathbf{c}_{\text{MMSE}} = \mathbf{R}^{-1}\hat{\mathbf{h}}_{n_0}. \tag{2.29}$$

Solving for the MMSE equalizer coefficients $\mathbf{c}_{\text{MMSE}}$ takes the form like the ZF equalizer coeffiencts in (2.19)

$$\mathbf{R}\mathbf{c}_{\text{MMSE}} = \hat{\mathbf{h}}_{n_0}. \tag{2.30}$$

The only difference between solving for the ZF and MMSE equalizer coefficients is $\mathbf{R}$ and $\mathbf{R}_{\hat{h}}$. The MMSE equalizer coefficients $\mathbf{c}_{\text{MMSE}}$ uses the noise variance estimate when building $\mathbf{R}$. The sparseness of $\mathbf{R}$ can also be leveraged to reduce computation drastically because $\mathbf{R}$ has the same sparse properties as $\mathbf{R}_{\hat{h}}$.

2.2.2 The Constant Modulus Algorithm

The bth CMA equalizer is an FIR filter defined by the coefficients

$$c_{\text{CMA}(b)}(-L_1) \quad \cdots \quad c_{\text{CMA}(b)}(0) \quad \cdots \quad c_{\text{CMA}(b)}(L_2). \tag{2.31}$$

The filter coefficients are calculated by a steepest decent algorithm

$$\mathbf{c}_{\text{CMA}(b+1)} = \mathbf{c}_{\text{CMA}(b)} - \mu \nabla J \tag{2.32}$$

initialized by the MMSE equalizer coefficients

$$\mathbf{c}_{\text{CMA}(0)} = \mathbf{c}_{\text{MMSE}}. \tag{2.33}$$

The vector $\mathbf{J}$ is the cost function and ∇J is the cost function gradient [8, eq. (352)]

$$\nabla J = \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \Big[y(n) y^*(n) - 1 \Big] \, y(n) \mathbf{r}^*(n). \tag{2.34}$$

where

$$\mathbf{r}(n) = \begin{bmatrix} r(n+L_1) \\ \vdots \\ r(n) \\ \vdots \\ r(n-L_2) \end{bmatrix}. \tag{2.35}$$

This means ∇J is defined by

$$\nabla J = \begin{bmatrix} \nabla J(-L_1) \\ \vdots \\ \nabla J(0) \\ \vdots \\ \nabla J(L_2) \end{bmatrix}. \tag{2.36}$$

A DSP engineer could implement the steepest decent algorithm by computing the cost function gradient directly. The L_{pkt} sample summation for ∇J in (2.34) does not map well to GPUs. Chapter 3 will show how well convolution performs in GPUs. The computation for ∇J can be massaged and re-expressed as convolution.

To begin messaging ∇J, the term

$$z(n) = 2\left[y(n)y^*(n) - 1\right] y(n) \tag{2.37}$$

is defined to simplify the expression of ∇J to

$$\nabla J = \frac{1}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} z(n)\mathbf{r}^*(n). \tag{2.38}$$

Expanding the expression of ∇J into vector form

$$\nabla J \;=\; \frac{z(0)}{L_{pkt}} \begin{bmatrix} r^*(L_1) \\ \vdots \\ r^*(0) \\ \vdots \\ r^*(L_2) \end{bmatrix} + \frac{z(1)}{L_{pkt}} \begin{bmatrix} r^*(1+L_1) \\ \vdots \\ r^*(1) \\ \vdots \\ r^*(1-L_2) \end{bmatrix} + \cdots \frac{z(L_{pkt}-1)}{L_{pkt}} \begin{bmatrix} r^*(L_{pkt}-1+L_1) \\ \vdots \\ r^*(L_{pkt}-1) \\ \vdots \\ r^*(L_{pkt}-1-L_2) \end{bmatrix} \tag{2.39}$$

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

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-k), \quad -L_1 \le k \le L_2. \tag{2.40}$$

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

$$\rho(n) = r^*(n). \tag{2.41}$$

Now

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)\rho(k-m). \tag{2.42}$$

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 $b(n)$ has support on $-L_{\text{pkt}}+1 \leq n \leq L_{\text{pkt}}-1$. Putting all the pieces together, we have

$$\begin{aligned} b(n) &= \sum_{m=0}^{L_{pkt}-1} z(m)\rho(n-m) \\ &= \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-n) \end{aligned} \tag{2.43}$$

Comparing Equation (2.42) and (2.43) shows that

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2. \tag{2.44}$$

The values of interest are shown in Figure 2.2.

This suggest the following matlab code for computing computing the gradient vector ∇J and implementing CMA.

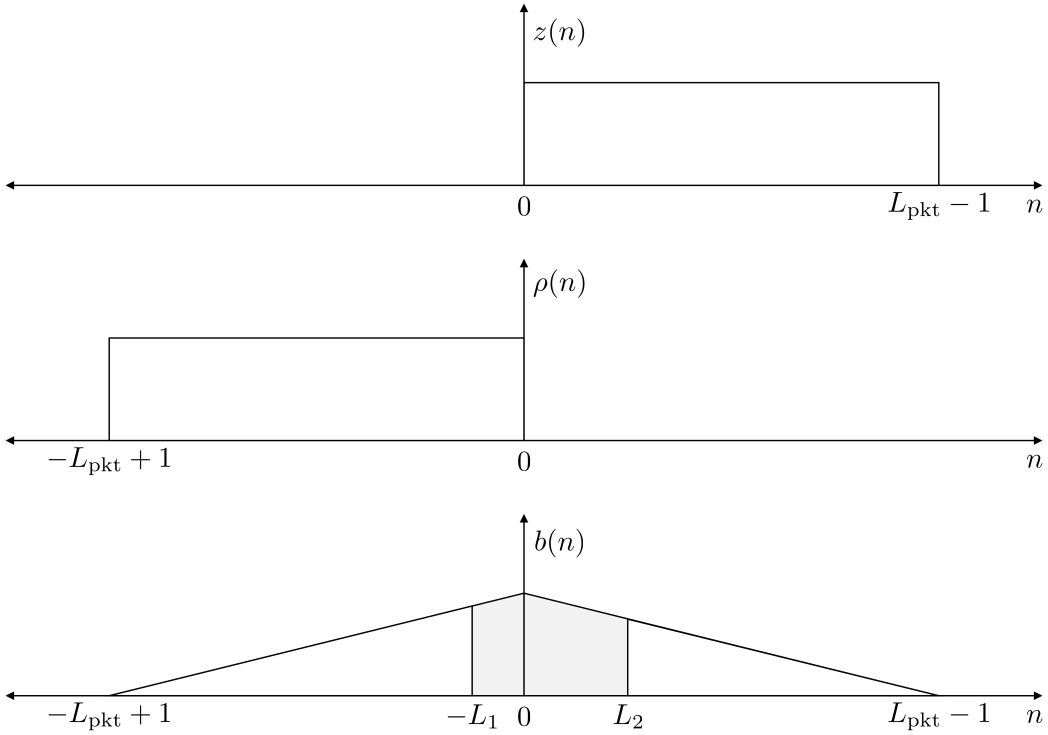


Figure 2.2: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.

Table 2.1: CMA

```
 1 c_CMA = c_MMSE;
 2 for i = 1:its
 3 yy = conv(r,c_CMAb);
 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 b = ifft(Z.*R);
 9 delJ = b(Lpkt-L1:Lpkt+L2);
10 c_CMAb1 = c_CMAb -mu*delJ;
11 c_CMAb = c_CMAb1;
12 end
13 yy = conv(r,c_CMA);
14 y = yy(L1+1:end-L2); % trim yy
```

2.2.3 The Frequency Domain Equalizers

Frequency Domain Equalizer One (FDE1) and Frequency Domain Equalizer Two (FDE2) are very similar and have the same structure. FDE1 and FDE2 are adapted from Williams and Saquib [9, eq. (11) and (12)].

Frequency Domain Equalizer One

FDE1 is the MMSE applied in the frequency domain from Williams' and Saquib's paper [9, eq. (11)].

$$C_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \cdots, L-1. \tag{2.45}$$

The term $C_{\text{FDE1}}(e^{j\omega_k})$ is FDE1s frequency response at ω_k. The term $\hat{H}(e^{j\omega_k})$ is the channel estimate frequency response at ω_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 (2.45) is straight forward to implement in GPUs. FDE1 is extremely fast and computationally efficient.

Frequency Domain Equalizer Two

FDE2 is also the MMSE or Wiener filter applied in the frequency domain but knowledge of the SOQPSK-TG spectrum is leveraged [9, eq. (12)]. The frequency respoonse of FDE2 is

$$C_{\text{FDE2}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \cdots, L-1 \tag{2.46}$$

where $\Psi(e^{j\omega_k})$ is the power spectral density of SOQPSK-TG shown in Figure 2.3. The term $\Psi(e^{j\omega_k})$ eliminates out of band multipath that may be challenging to estimate and over come.

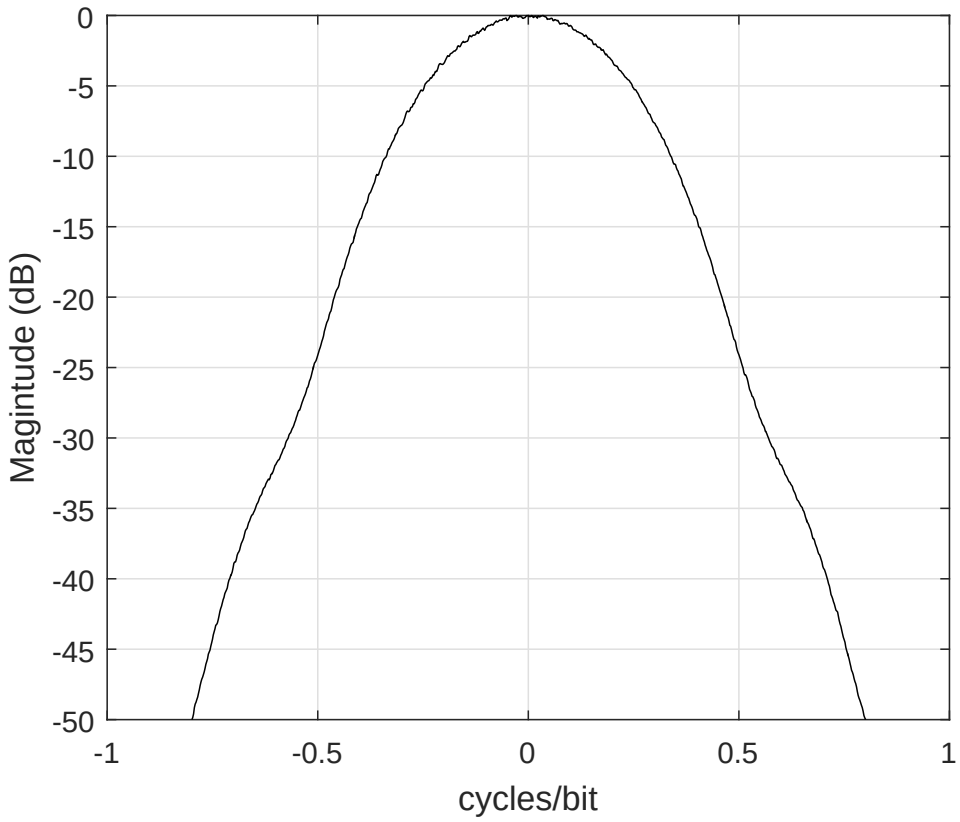


Figure 2.3: SOQPSK-TG power spectral density.

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. Telsa GPUs are built for general purpose high performance computing. Figure 3.1 shows the form factor of a Tesla K40c and K20c. In 2007 NVIDIA released an extension to C, C++ and Fortran called CUDA (Compute Unified Device Architecture).

CUDA enables GPUs to be used for high performance computing in computer vision, deep learning, artificial intelligence and signal processing [11]. 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:

- Optimizing memory access leads to faster execution time rather than optimizing number of floating point operations.
- The number of threads per block can majorly affect execution time.
- CPU and GPU processing can be pipelined.
- Convolution maps very well to GPUs using the Fast Fourier Transform.
- 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 CPU verse CUDA example

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:

$$\begin{aligned} \mathbf{C}_1 &= \mathbf{A}_1 + \mathbf{B}_1 \\ \mathbf{C}_2 &= \mathbf{A}_2 + \mathbf{B}_2 \end{aligned} \tag{3.1}$$

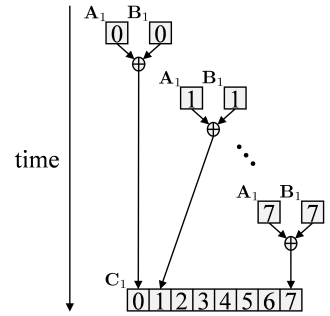


Figure 3.2: A block diagram of how a CPU sequentially performs vector addition.

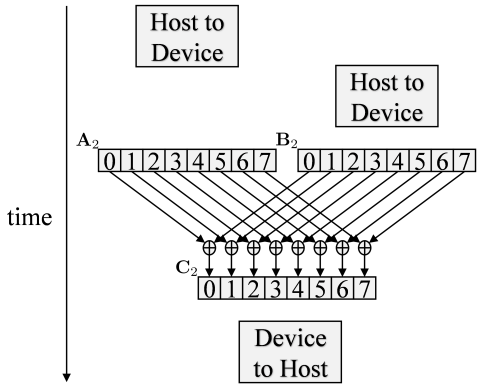


Figure 3.3: A block diagram of how a GPU performs vector addition in parallel.

where $\mathbf{C}_1$ is computed in the CPU and $\mathbf{C}_2$ is computed in the GPU. Line 42 the CPU computes $\mathbf{C}_1$ by summing elements of $\mathbf{A}_1$ and $\mathbf{B}_1$ together *sequentially*. Figure 3.2 shows how the CPU computes $\mathbf{C}_1$ sequentially.

The vector addition in the GPU takes a little more work. On lines 60 and 61 the vectors in host memory $\mathbf{A}_1$ and $\mathbf{B}_1$ are copied to device memory vectors $\mathbf{A}_2$ and $\mathbf{B}_2$. The vector $\mathbf{C}_2$ is computed by calling the GPU kernel VecAddGPU on line 75. The vector 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.

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

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

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

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

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

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

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

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

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

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

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

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

        // 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 index called threadIdx and blockIdx. threadIdx is the thread index inside the assigned thread block. blockIdx is the index of the block that the thread is assigned to. Both threadIdx and blockIdx are three dimensional and have x, y and z components. In this thesis only the x dimension is used because GPU kernels operate only on one dimensional vectors. blockDim is the number of threads assigned per block, in fact blockDim is equal to the number of threads per block because the vectors are one dimensional.

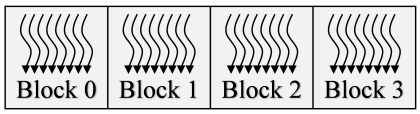


Figure 3.4: 32 threads launched in 4 thread blocks with 8 threads per block.

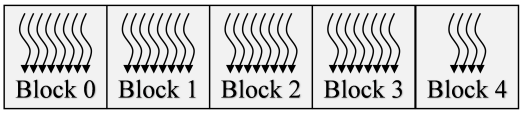


Figure 3.5: 36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.

To turn CPU the N long "for loop" on line 7 into 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 will 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 but only kilobytes are available. The slowest memory is public memory in the form of L2 cache and global memory and gigabytes 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

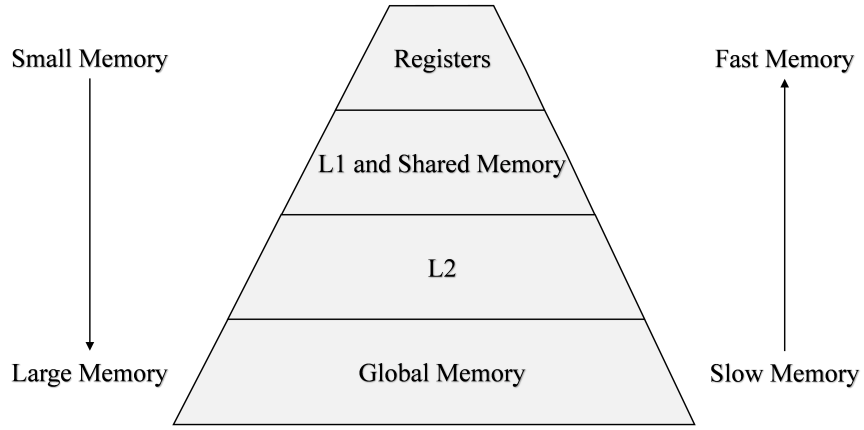


Figure 3.6: Diagram comparing memory size and speed. Global memory is massive but extremely slow. Registers are extremely fast but there are very few.

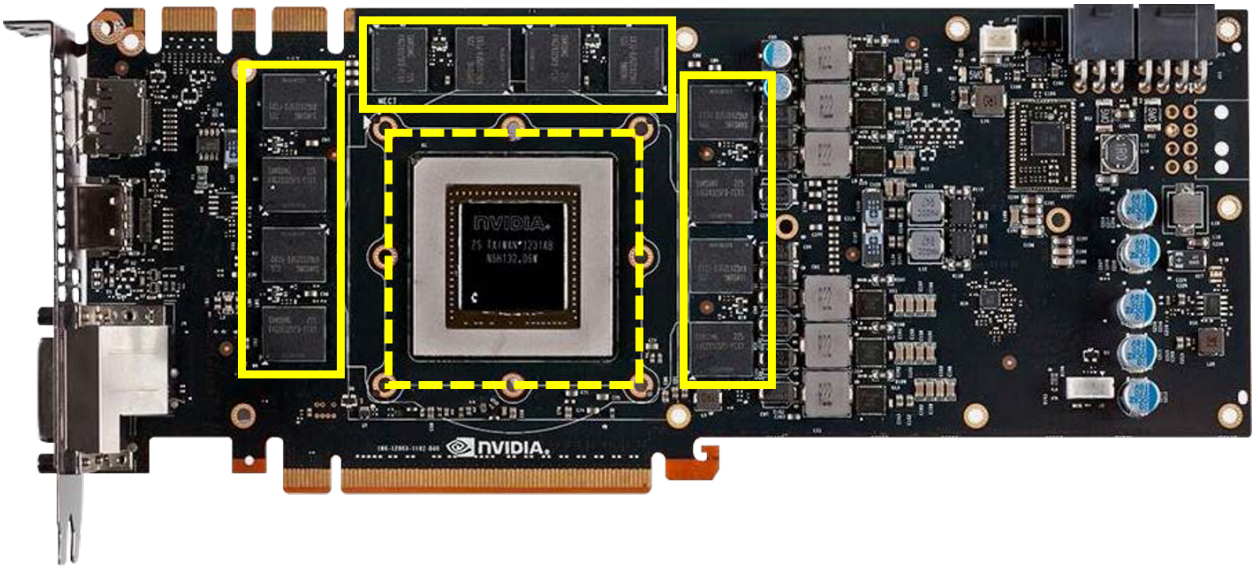


Figure 3.7: Example of an NVIDIA GPU card. The GPU chip with registers, L1 cache and shared memory is shown in the dashed box. The L2 cache and global memory is shown off chip in the solid boxes.

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

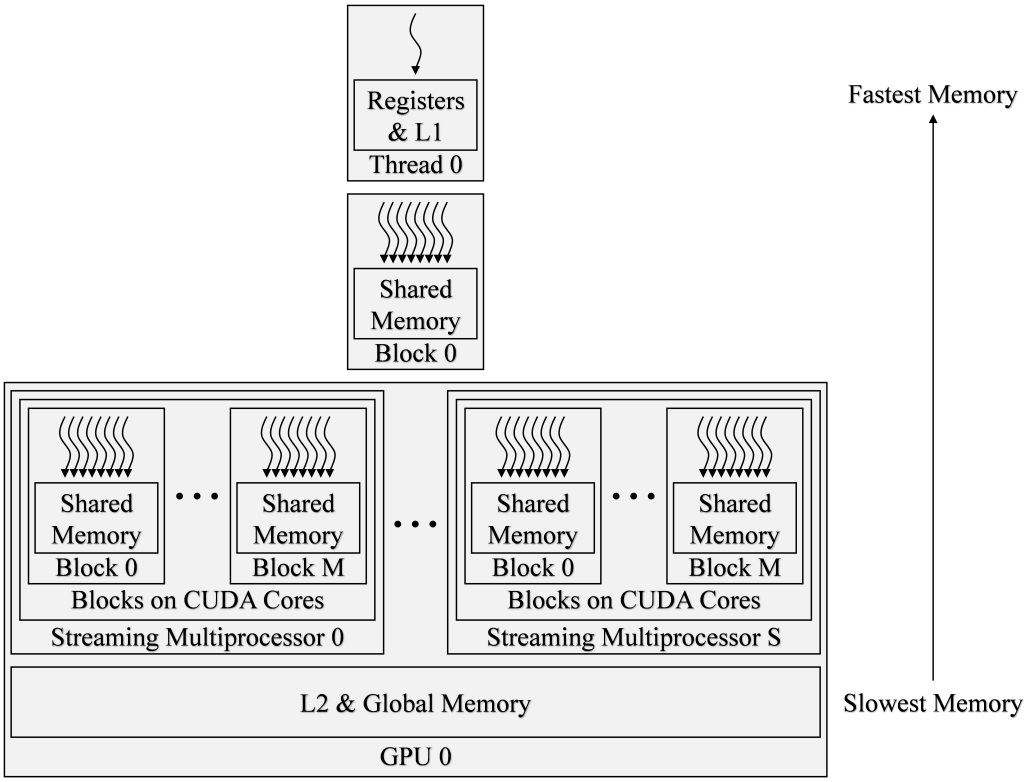


Figure 3.8: A block diagram where local, shared, and global memory is located. Each thread has private local memory. Each thread block has private shared memory. The GPU has global memory that all threads can access.

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 Telsa K40c and K20c resources.

3.1.4 Thread Optimization

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

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

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. Luckily, there is a finite number of possible threads per block in the Tesla K40c and K20c GPUs, 1 to 1024. Listing 3.2 shows a simple test program that times GPU kernel execution time while sweeping 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.

```
float milliseconds_opt = pow(2,10); // initiaize to "big" number
int T_B_opt;
int minNumTotalThreads = pow(2,20); // set to minimum number of required threads
for(int T_B = 1; T_B<=1024; T_B++){
        int B = minNumTotalThreads/T_B;
        if(minNumTotalThreads % T_B > 0)
                B++;
        cudaEvent_t start, stop;
        cudaEventCreate(&start);
        cudaEventCreate(&stop);
        cudaEventRecord(start);

        GPUkernel<<<B, T_B>>>(dev_vec0, dev_vec1);

        cudaEventRecord(stop);
        cudaEventSynchronize(stop);
        float milliseconds = 0;
        cudaEventElapsedTime(&milliseconds, start, stop);
        cudaEventDestroy(start);
        cudaEventDestroy(stop);
        if(milliseconds<milliseconds_opt){
                milliseconds_opt = milliseconds;
                T_B_opt = T_B;
        }
}
cout << "Optimal Threads Per Block " << T_B_opt << endl
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 do computations in *warps*. Warps are groups of 32 threads that do every computation together in lock step. If the number of threads per block is a non multiple of 32, some threads in a warp will be idle and the GPU will have 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 with exceptional documentation that are extremely useful and efficient. The CUDA libraries are written by NVIDIA engineers that know how to squeeze out every drop

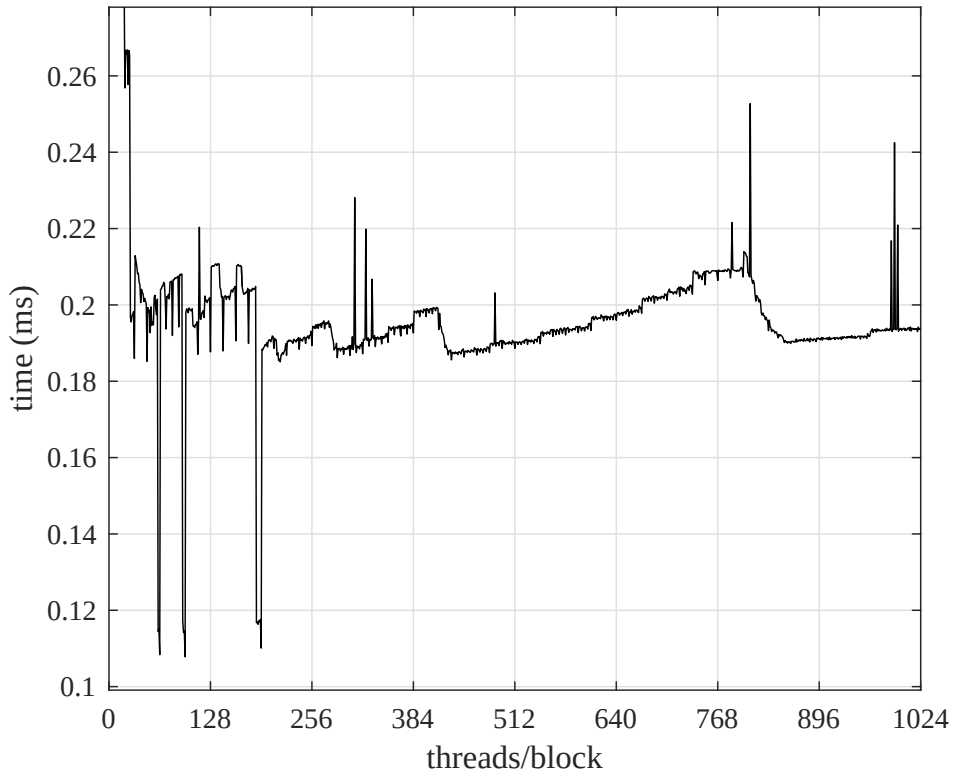


Figure 3.9: Plot showing how execution time is affected by changing the number of threads per block. The optimal execution time for an example GPU kernel is 0.1078 ms at the optimal 96 threads per block.

of performance out of NVIDIA GPUs. Some libraries used in this thesis are cuFFT, cuBLAS and cuSolverSp.

3.1.5 CPU and GPU Pipelining

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

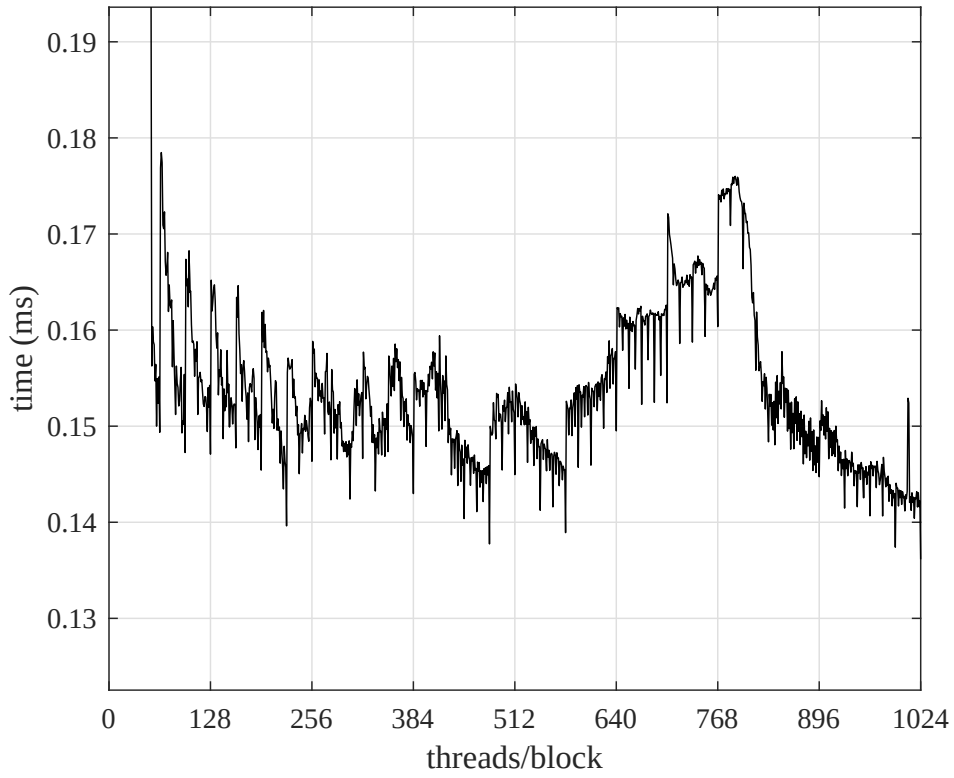


Figure 3.10: Plot showing the number of threads per block doesn't always drastically affect execution time.

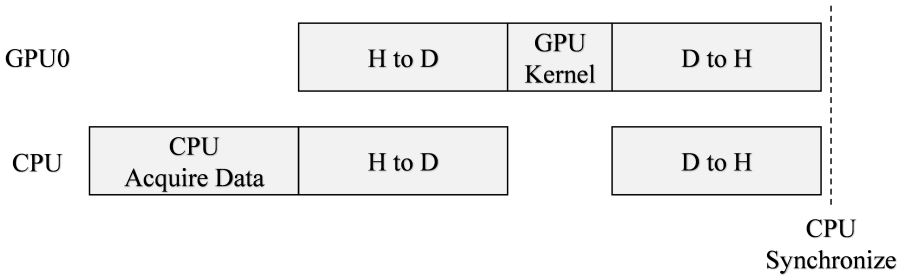


Figure 3.11: The typical approach of CPU and GPU operations. This block diagram shows the profile of Listing 3.3.

blocks CPU until all GPU instructions are finished executing. Note that the CPU is blocked during any host to device or device to host transfer. Acquiring and copying data takes processing time on the CPU and GPU. Figure 3.12 shows a block diagram of what is happening on the CPU and GPU in Listing 3.3. The GPU is idle while the CPU is acquiring data and the CPU is idle while the GPU is processing.

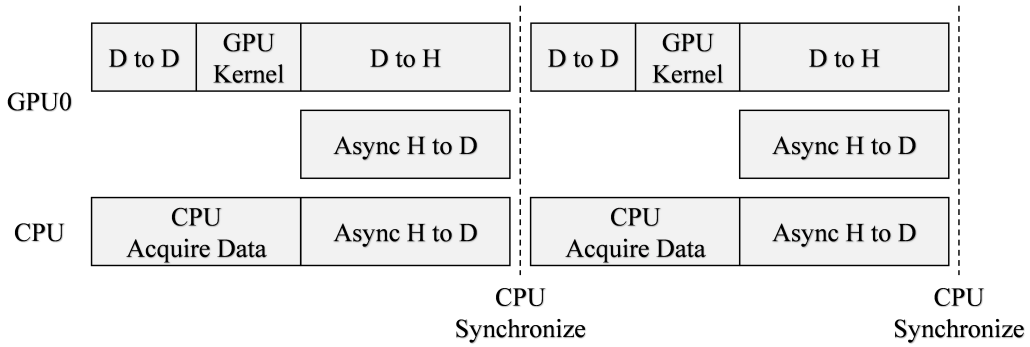


Figure 3.12: GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 3.4.

Listing 3.4 shows how to CPU and GPU operations can be pipelined. Assuming data is already on the GPU from a prior iteration, 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.

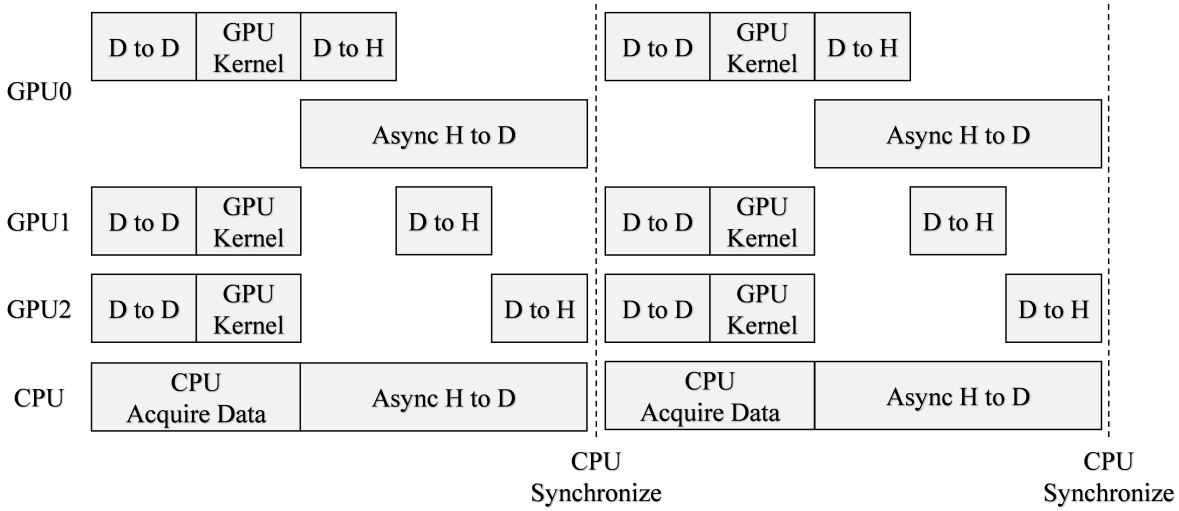


Figure 3.13: A block diagram of pipelining a CPU with three GPUs.

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.

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

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

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

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

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

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

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

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

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

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

3.2 GPU Convolution

Convolution is one of the most important tools in digital signal processing. The PAQ system explained in Chapter 2 uses convolution up to 26 times per packet, depending on the number of CMA iterations. If convolution execution time can be reduced by 10 ms, the full system execution time can reduced by 260 ms.

Discrete time convolution applies the L sample complex filter $\mathbf{h}$ to the N sample complex signal $\mathbf{x}$. The filtered signal $\mathbf{y}$ is $C = L + N - 1$ complex samples long. Convolution in the time domain is

$$y(n) = \sum_{m=0}^{L-1} x(m)h(n-m) \tag{3.2}$$

and the frequency domain is

$$\mathbf{y} = \mathscr{F}^{-1}(\mathscr{F}(\mathbf{x}) \times \mathscr{F}(\mathbf{h})). \tag{3.3}$$

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.
- 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.

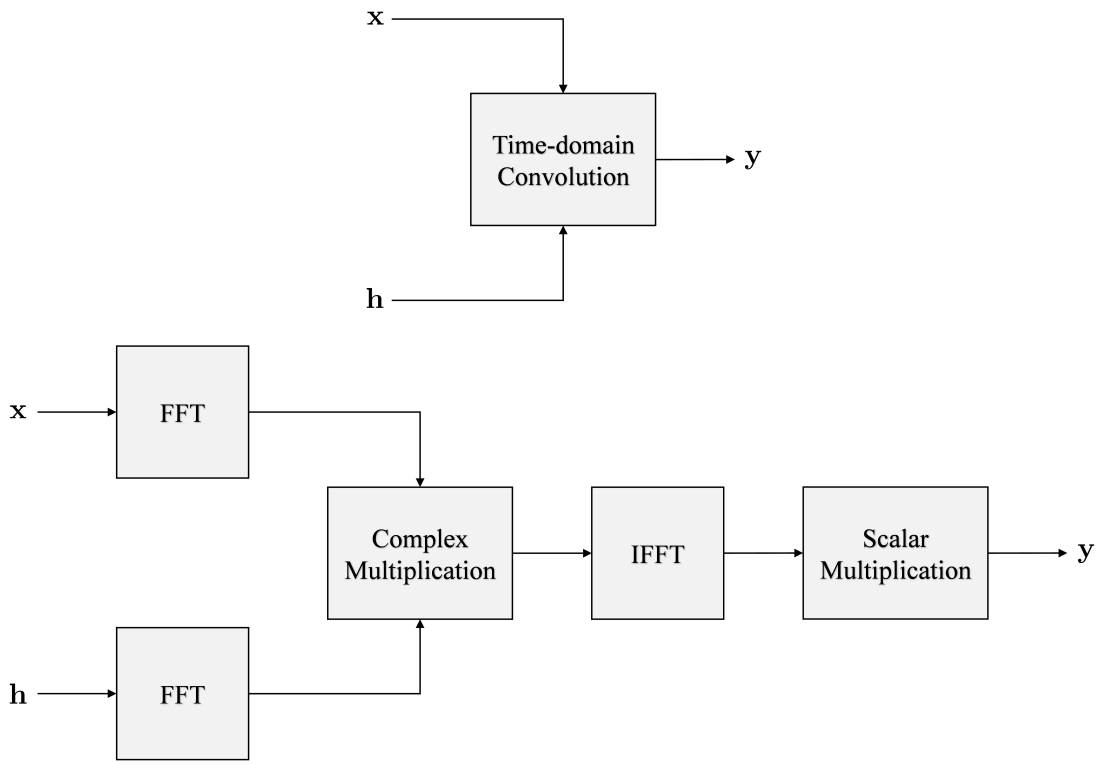


Figure 3.14: Block diagrams showing showing time-domain convolution and frequency-domain convolution.

3.2.1 Floating Point Operation Comparison

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

$$(A + jB) \times (C + jD) = (AC - BD) + j(AD + BC) \tag{3.4}$$

requires 6 flops, 4 multiplications and 2 additions/subtractions. Output elements of $\mathbf{y}$ in Equation (3.2) requires $8L = (6+2)L$ flops, 2 extra flops are required for each summand. The time-domain convolution requires

$$8LC \text{ flops} \tag{3.5}$$

where $C = N + L - 1$ is the number of samples in the filtered signal.

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 is the next power of 2 above C. Both the CPU based Fastest Fourier Transform in the West (FFTW) library and the NVIDIA GPU cuFFT library use the Cooley-Tukey radix 2 transform. Each forward or backward Fourier transform requires $5M\log_2(M)$ flops [12, 13]. As shown by Equation (3.3), frequency-domain convolution requires

$$3 \times 5M\log_2(M) + 6M \text{ flops} \tag{3.6}$$

from 3 FFTs and a M point to point multiplication.

Sections 1.2 and 2.1.5 show the PAQ system has one signal length $N = L_{\text{pkt}} = 12672$ 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 Batch Convolution Comparison

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

- time-domain convolution in a CPU
- frequency-domain convolution in a CPU
- time-domain convolution in a GPU using global memory
- time-domain convolution in a GPU using shared memory

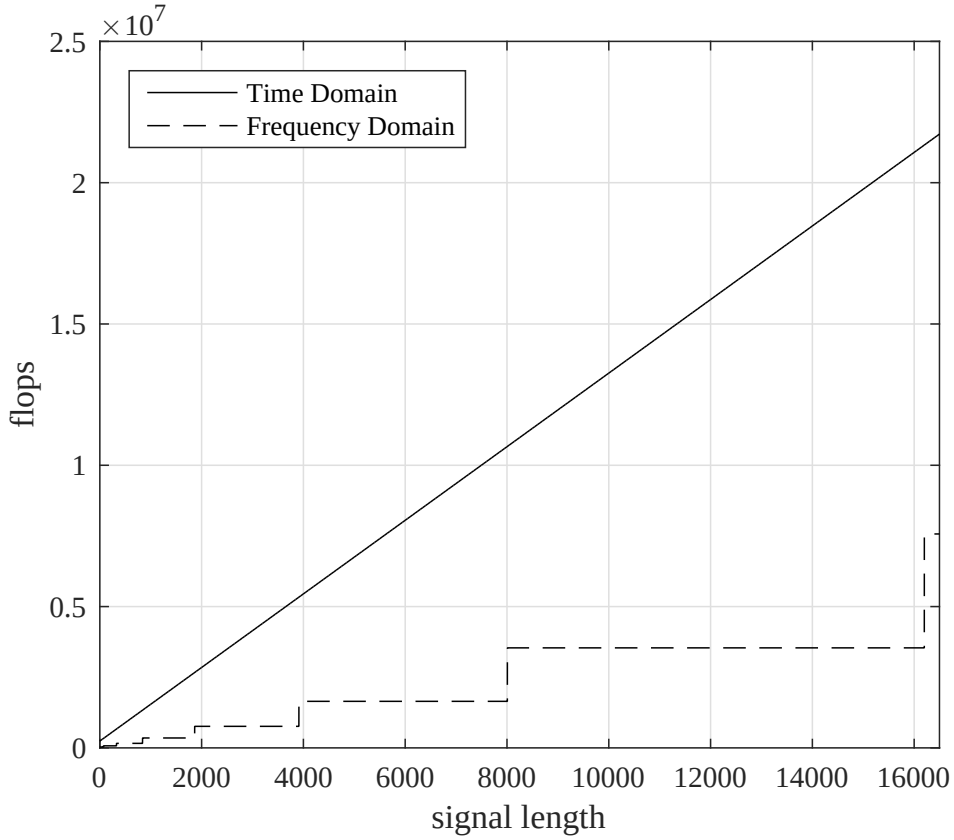


Figure 3.15: Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 186 tap complex filter.

- frequency-domain convolution in a GPU

The three time-domain convolution implementations compute (3.2) directly. The two frequency-domain convolution implementations compute (3.3) using the CPU FFTW library and the GPU based cuFFT library. For a given signal and filter length, a good CUDA programmer can make an educated guess on which algorithm may be 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

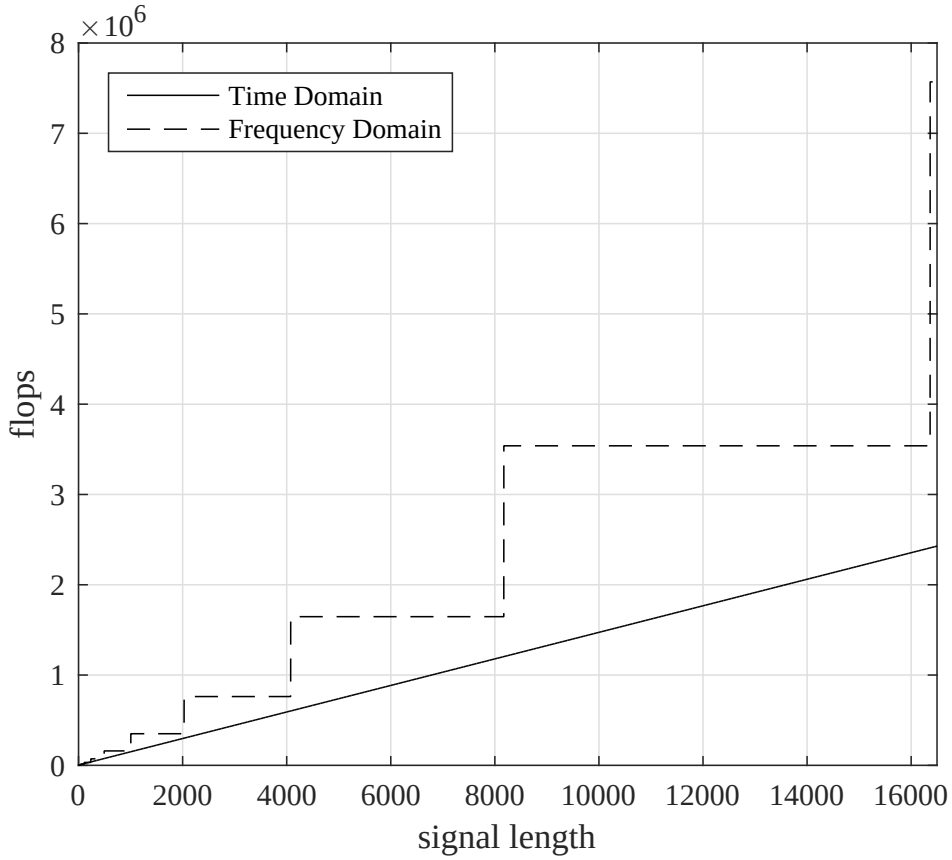


Figure 3.16: Comparison of number of floating point operations (flops) required to convolve a variable length complex signal with a 21 tap complex filter.

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

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

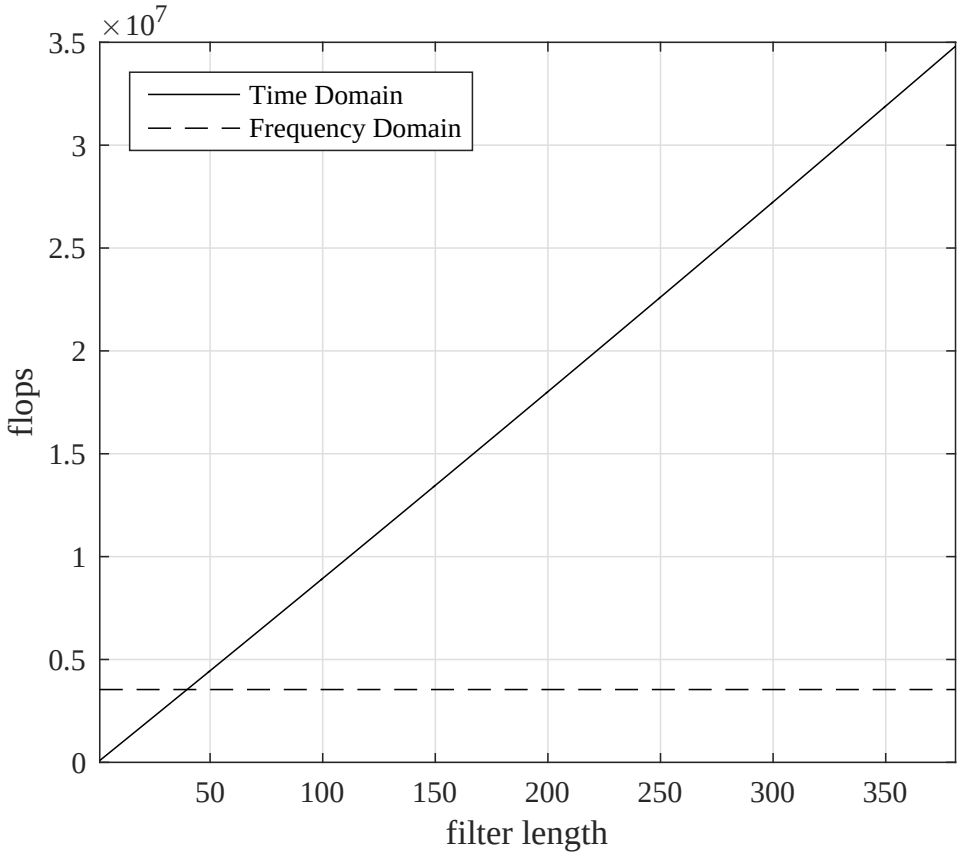


Figure 3.17: Comparison of number of floating point operations (flops) required to convolve a 12672 sample complex signal with a variable length tap complex filter.

emphasize points that are of interest to the PAQ system. The noise in Figure 3.18 stems from the host operating system context switching. To clean up the time samples, local minimums were found in windows ranging from 3 to 15 samples.

Comparing Figures 3.19 through 3.21 to Figures 3.15 through 3.17 shows CPU convolution has the structure that the number of flops predicted but GPU does not. The convolution execution time comparison reinforces 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 PAQ signal and filter lengths when performing a single complex convolution.

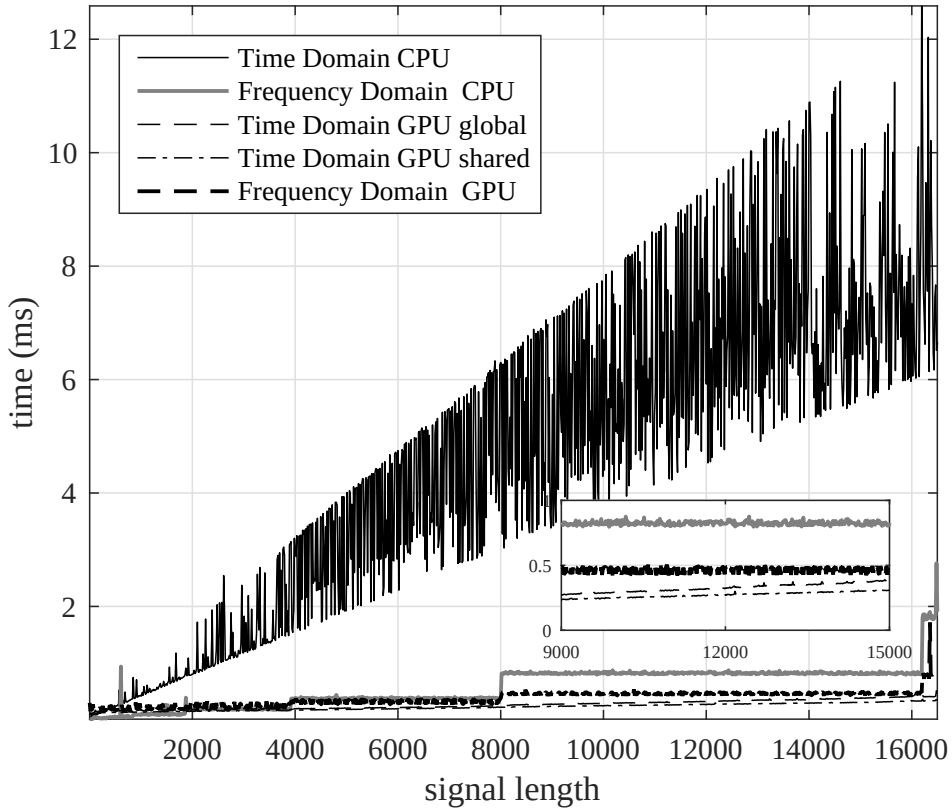


Figure 3.18: Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 186 taps. The comparison is messy with out lower bounding.

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

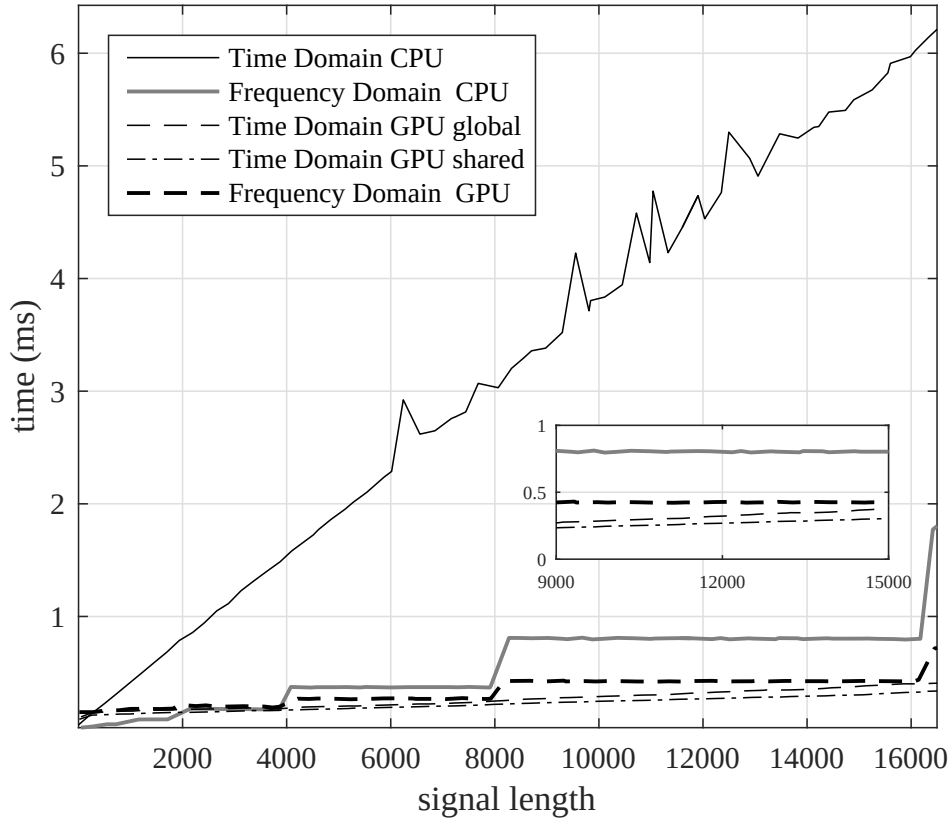


Figure 3.19: Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 186 taps. A lower bound was applied by searching for a local minimums in 15 sample width windows.

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	ConvGPUshared	0.1971
GPU frequency domain	cuFFT	0.3360

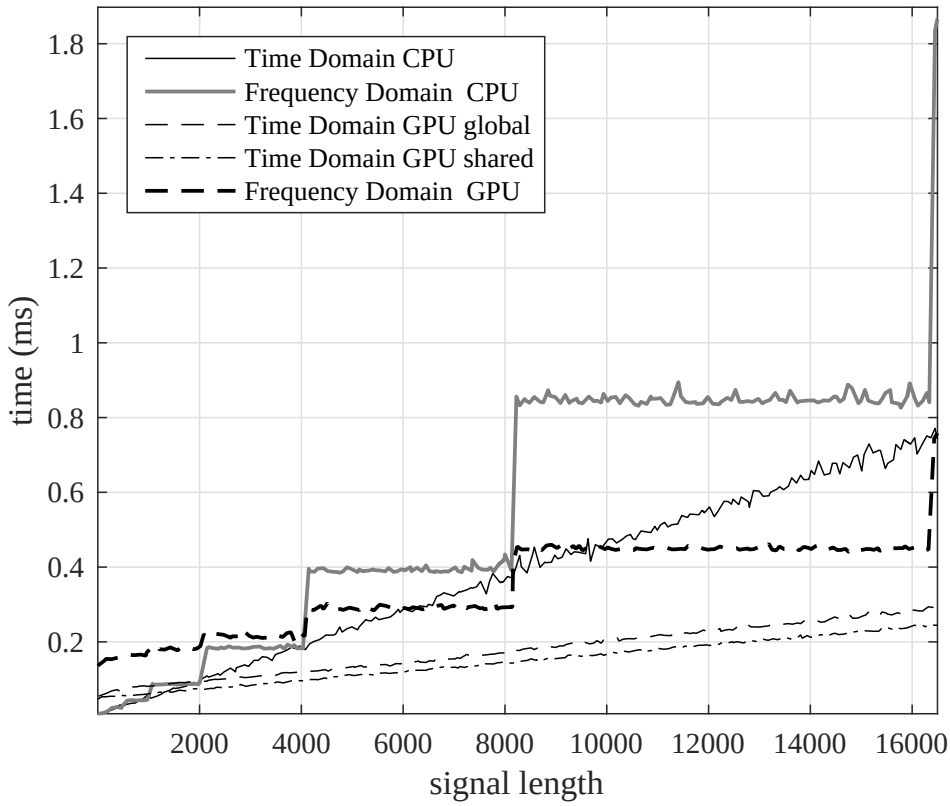


Figure 3.20: Comparison of a complex convolution on CPU verse GPU. The signal length is variable and the filter is fixed at 21 taps. A lower bound was applied by searching for a local minimums in 5 sample width windows.

3.2.3 Batched Convolution

Section 3.2.2 illustrated single convolution doesn't leverage the full power of parallel processing in GPUs. The PAQ project received signal has a packetized structure with 3104 packets per 1907 ms. Rather than processing each packet separately, many packets are buffered then 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. showed batched libaries achive more Gflops than calling GPU kernels multiple times [14]. Listing 3.6 (at the end of the chapter) shows three GPU implementations of batch processing convolution and Table 3.5 shows how the execution time of the code was measured.

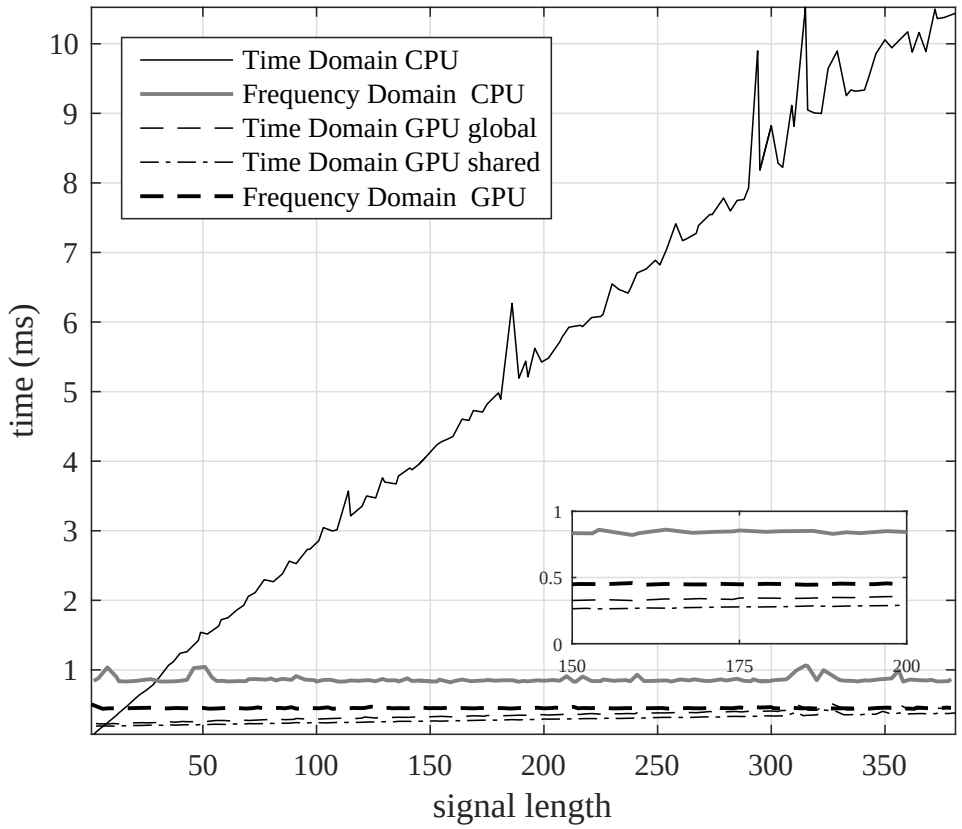


Figure 3.21: Comparison of a complex convolution on CPU verse GPU. The filter length is variable and the signal is fixed at 12672 samples. A lower bound was applied by searching for a local minimums in 3 sample width windows.

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

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

Figure 3.22 compares execution time of batch processing convolution 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 batch convolution is not competitive with GPU batch convolution and thus CPU batched processing was not explored any further.

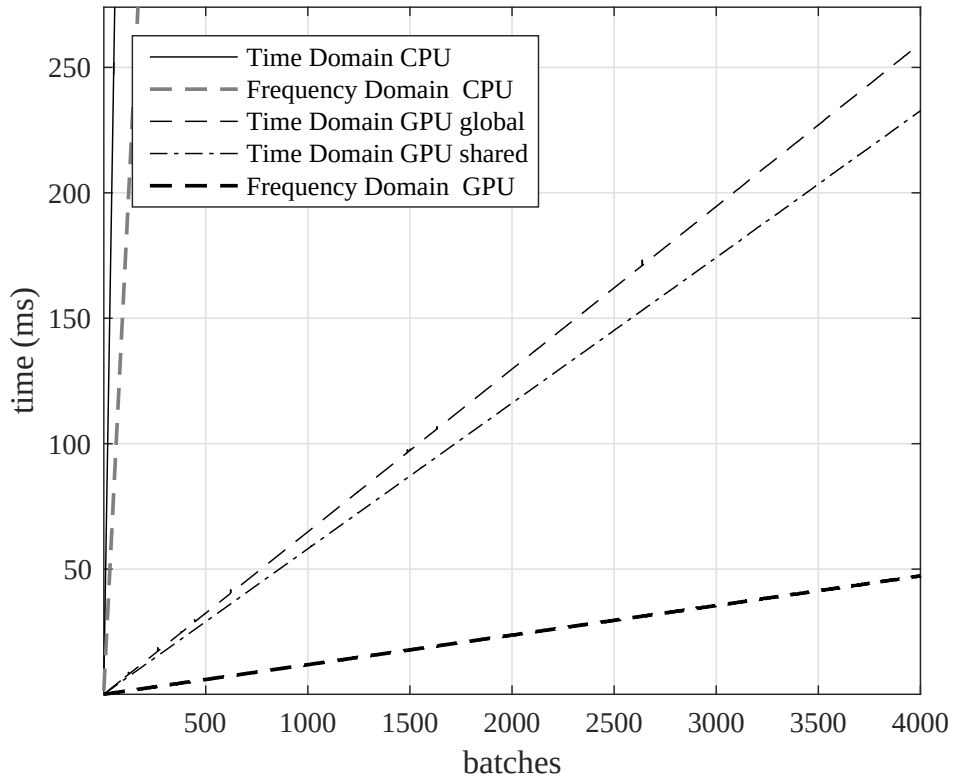


Figure 3.22: Comparison of a batched complex convolution on a CPU and GPU. The number of batches is variable while the signal and filter length is set to 12672 and 186.

Now that the GPU and CPU execution time is not being compared, Table 3.5 shows execution times include only GPU kernels and exclude memory transfers. Figure 3.23 compares GPU batch convolution 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 PAQ signal and filter lengths when performing batched convolution. The fastest batch implementation execution time of convolving a 12672 sample signal with a filter depends on the number of taps. Batch frequency-domain convolution is fastest for 186 tap filters while time-domain convolution using shared memory is fastest for 21 tap filters.

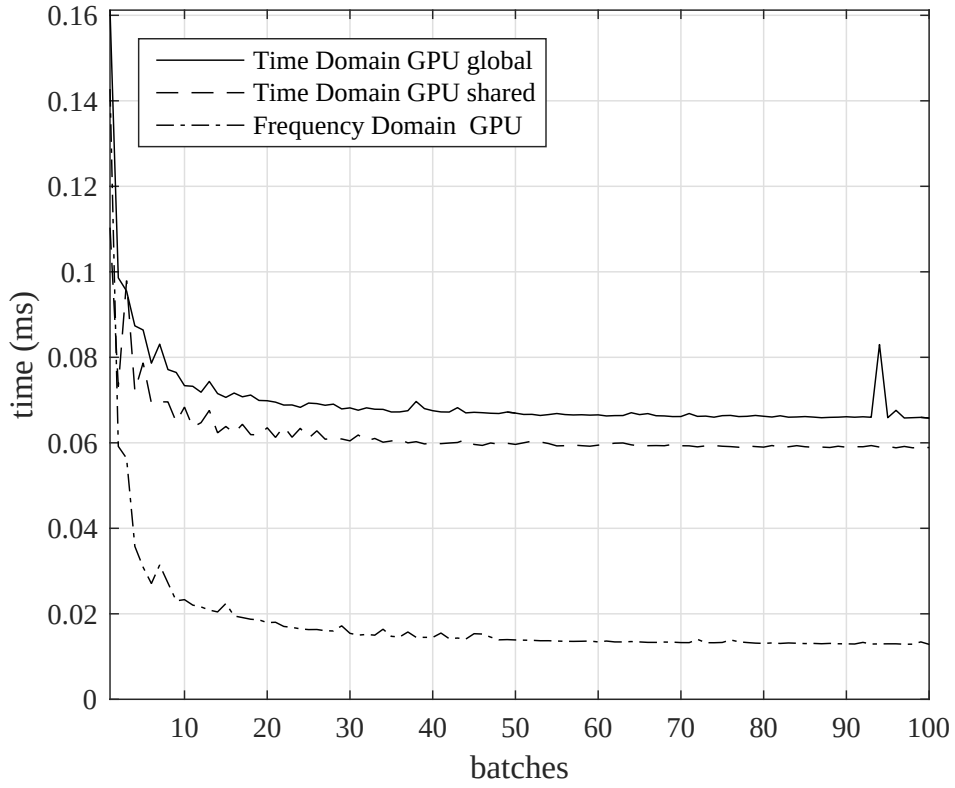


Figure 3.23: Comparison on execution time per batch for complex convolution. The number of batches is variable while the signal and filter length is set to 12672 and 186.

Table 3.6: Batched convolution 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	ConvGPUshared	180.272
GPU frequency domain	cuFFT	36.798

Table 3.7: Batched convolution 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	ConvGPUshared	20.4287
GPU frequency domain	cuFFT	36.7604

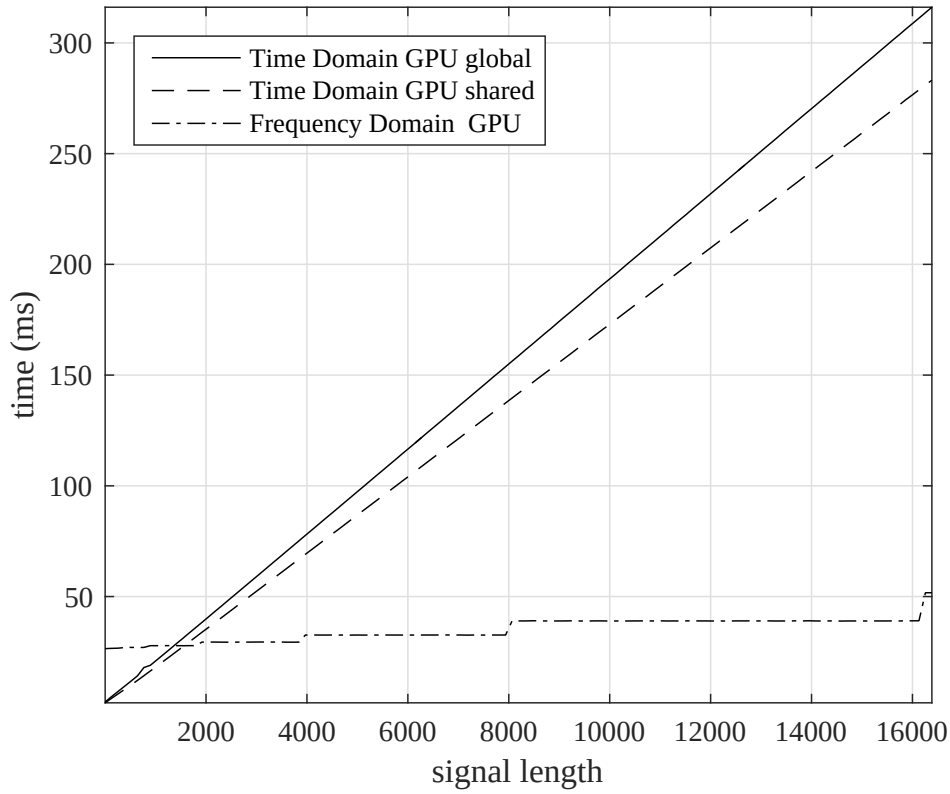


Figure 3.24: Comparison of a batched complex convolution on a GPU. The signal length is variable and the filter is fixed at 186 taps.

Until now, convolving one signal with only one filter has been considered. In section 1.2, Figure 1.8 showed the PAQ system the signal is filtered by two cascaded filters: an equalizer filter and a detection filter. The block diagrams in Figure 3.27 show steps required for cascading time-domain and frequency-domain convolution.

Comparing the block diagrams in Figures 3.27 and 3.14, cascading filters in the time domain requires two time-domain convolutions while cascading two filters in the frequency domain requires an extra FFT and complex multiplication. The second time-domain convolution is applying a 206 tap filter to a 12672 sample signal because the output of the first time-domain convolution is $206 = 21 + 186 - 1$ samples long because the detection filter is 21 taps and the equalizer filter is 186 taps. Table 3.8 shows the execution times for PAQ signal and filter lengths when performing batched cascaded convolution. Frequency-domain batched cascaded convolution is fastest for the PAQ system.

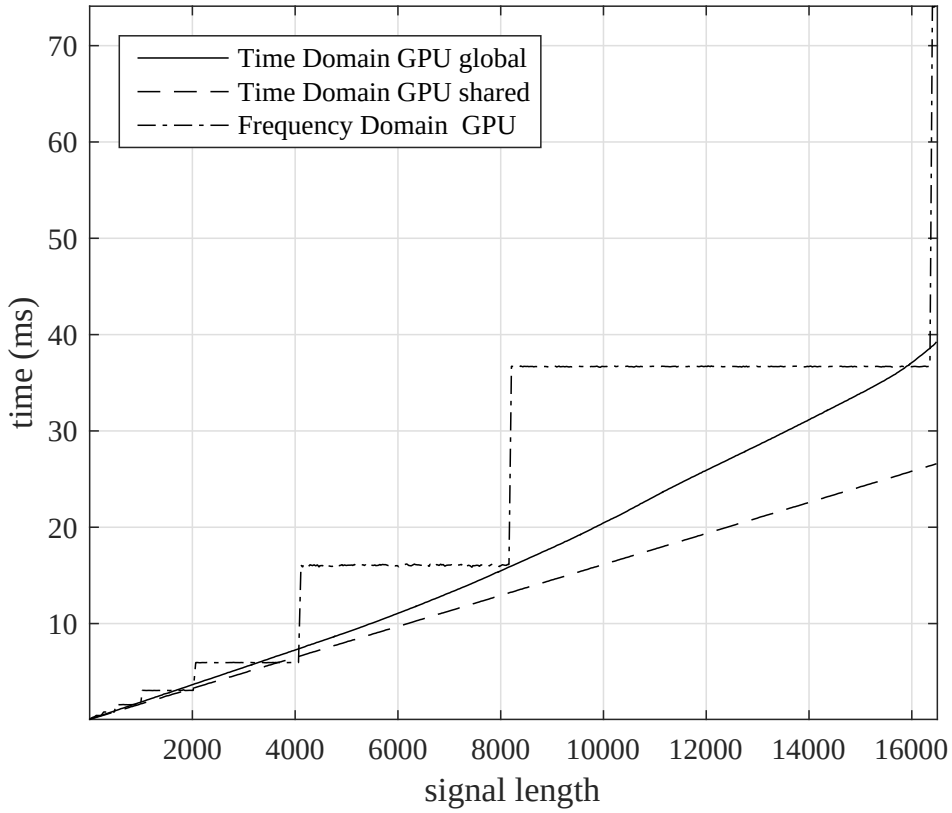


Figure 3.25: Comparison of a batched complex convolution on a GPU. The signal length is variable and the filter is fixed at 21 taps.

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	ConvGPUshared	200.018
GPU frequency domain	cuFFT	39.0769

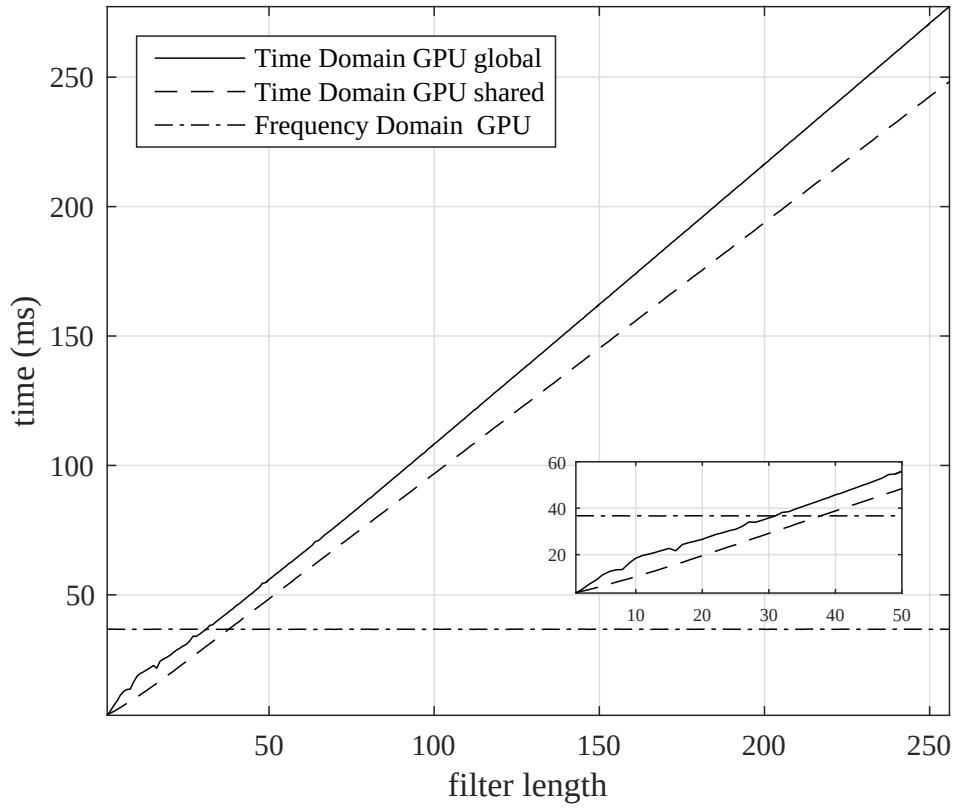


Figure 3.26: Comparison of a batched complex convolution on a GPU. The filter length is variable and the signal length is set to 12672 samples.

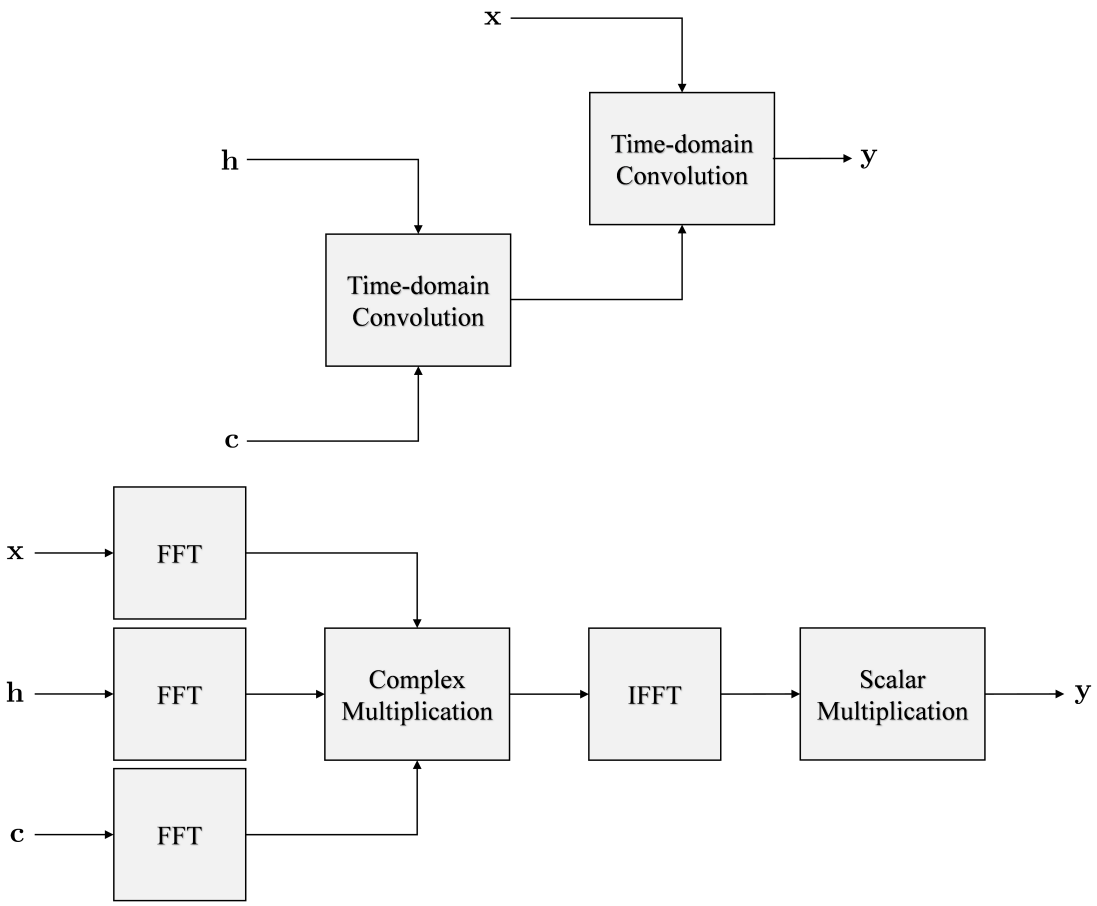


Figure 3.27: Block diagrams showing showing cascaded time-domain convolution and frequency-domain convolution.

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

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


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

}

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

        int lastThread = Lx+Lh-1;

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

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

```
65
66
67  __global__ void ConvGPUshared(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
68          int yIdx = blockIdx.x*blockDim.x + threadIdx.x;
69
70          int lastThread = Lx+Lh-1;
71
72          extern __shared__ cufftComplex h_shared[];
73          if(threadIdx.x < Lh){
74                  h_shared[threadIdx.x] = h[threadIdx.x];
75          }
76          __syncthreads();
77
78          // don't access elements out of bounds
79          if(yIdx >= lastThread)
80                  return;
81
82          cufftComplex temp;
83          temp.x = 0;
84          temp.y = 0;
85          for(int hIdx = 0; hIdx < Lh; hIdx++){
86                  int xAccessIdx = yIdx-hIdx;
87                  if(xAccessIdx>=0 && xAccessIdx<Lx){
88                          // temp += x[xAccessIdx]*h[hIdx];
89                          float A = x[xAccessIdx].x;
90                          float B = x[xAccessIdx].y;
91                          float C = h_shared[hIdx].x;
92                          float D = h_shared[hIdx].y;
93                          cufftComplex result;
94                          result.x = A*C-B*D;
95                          result.y = A*D+B*C;
96                          temp.x += result.x;
97                          temp.y += result.y;
98                  }
99          }
100         y[yIdx] = temp;
101 }
102
103 __global__ void PointToPointMultiply(cufftComplex* v0, cufftComplex* v1, int lastThread){
104         int i = blockIdx.x*blockDim.x + threadIdx.x;
105
106         // don't access elements out of bounds
107         if(i >= lastThread)
108                 return;
109         float A = v0[i].x;
110         float B = v0[i].y;
111         float C = v1[i].x;
112         float D = v1[i].y;
113
114         // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
115         cufftComplex result;
116         result.x = A*C-B*D;
117         result.y = A*D+B*C;
118
119         v0[i] = result;
120 }
121
122 __global__ void ScalarMultiply(cufftComplex* vec0, float scalar, int lastThread){
123         int i = blockIdx.x*blockDim.x + threadIdx.x;
124
125         // Don't access elements out of bounds
126         if(i >= lastThread)
127                 return;
128         cufftComplex scalarMult;
129         scalarMult.x = vec0[i].x*scalar;
130         scalarMult.y = vec0[i].y*scalar;
131         vec0[i] = scalarMult;
132 }
```

```
133
134 int main(){
135         int N = 1000;
136         int L = 186;
137         int C = N + L - 1;
138         int M = pow(2, ceil(log(C)/log(2)));
139
140         cufftComplex *mySignal1;
141         cufftComplex *mySignal2;
142         cufftComplex *mySignal2_fft;
143
144         cufftComplex *myFilter1;
145         cufftComplex *myFilter2;
146         cufftComplex *myFilter2_fft;
147
148         cufftComplex *myConv1;
149         cufftComplex *myConv2;
150         cufftComplex *myConv2_timeReversed;
151         cufftComplex *myConv3;
152         cufftComplex *myConv4;
153         cufftComplex *myConv5;
154
155         mySignal1               = (cufftComplex*)malloc(N*sizeof(cufftComplex));
156         mySignal2               = (cufftComplex*)malloc(M*sizeof(cufftComplex));
157         mySignal2_fft           = (cufftComplex*)malloc(M*sizeof(cufftComplex));
158
159         myFilter1               = (cufftComplex*)malloc(L*sizeof(cufftComplex));
160         myFilter2               = (cufftComplex*)malloc(M*sizeof(cufftComplex));
161         myFilter2_fft           = (cufftComplex*)malloc(M*sizeof(cufftComplex));
162
163         myConv1                 = (cufftComplex*)malloc(C*sizeof(cufftComplex));
164         myConv2                 = (cufftComplex*)malloc(M*sizeof(cufftComplex));
165         myConv2_timeReversed= (cufftComplex*)malloc(M*sizeof(cufftComplex));
166         myConv3                 = (cufftComplex*)malloc(C*sizeof(cufftComplex));
167         myConv4                 = (cufftComplex*)malloc(C*sizeof(cufftComplex));
168         myConv5                 = (cufftComplex*)malloc(M*sizeof(cufftComplex));
169
170         srand(time(0));
171         for(int i = 0; i < N; i++){
172                 mySignal1[i].x = rand()%100-50;
173                 mySignal1[i].y = rand()%100-50;
174         }
175
176         for(int i = 0; i < L; i++){
177                 myFilter1[i].x = rand()%100-50;
178                 myFilter1[i].y = rand()%100-50;
179         }
180
181         cufftComplex *dev_mySignal3;
182         cufftComplex *dev_mySignal4;
183         cufftComplex *dev_mySignal5;
184
185         cufftComplex *dev_myFilter3;
186         cufftComplex *dev_myFilter4;
187         cufftComplex *dev_myFilter5;
188
189         cufftComplex *dev_myConv3;
190         cufftComplex *dev_myConv4;
191         cufftComplex *dev_myConv5;
192
193         cudaMalloc(&dev_mySignal3, N*sizeof(cufftComplex));
194         cudaMalloc(&dev_mySignal4, N*sizeof(cufftComplex));
195         cudaMalloc(&dev_mySignal5, M*sizeof(cufftComplex));
196
197         cudaMalloc(&dev_myFilter3, L*sizeof(cufftComplex));
198         cudaMalloc(&dev_myFilter4, L*sizeof(cufftComplex));
199         cudaMalloc(&dev_myFilter5, M*sizeof(cufftComplex));
200
```

```
201         cudaMalloc(&dev_myConv3,   C*sizeof(cufftComplex));
202         cudaMalloc(&dev_myConv4,   C*sizeof(cufftComplex));
203         cudaMalloc(&dev_myConv5,   M*sizeof(cufftComplex));
204
205
206         /**
207          * Time-domain Convolution CPU
208          */
209         ConvCPU(myConv1,mySignal1,myFilter1,N,L);
210
211         /**
212          * Frequency Domain Convolution CPU
213          */
214         fftwf_plan forwardPlanSignal = fftwf_plan_dft_1d(M, (fftwf_complex*)mySignal2,    (
                fftwf_complex*)mySignal2_fft,           FFTW_FORWARD, FFTW_MEASURE);
215         fftwf_plan forwardPlanFilter = fftwf_plan_dft_1d(M, (fftwf_complex*)myFilter2,   (
                fftwf_complex*)myFilter2_fft,           FFTW_FORWARD, FFTW_MEASURE);
216         fftwf_plan backwardPlanConv  = fftwf_plan_dft_1d(M, (fftwf_complex*)mySignal2_fft,(
                fftwf_complex*)myConv2_timeReversed, FFTW_FORWARD, FFTW_MEASURE);
217
218         cufftComplex zero; zero.x = 0; zero.y = 0;
219         for(int i = 0; i < M; i++){
220                 if(i<N)
221                         mySignal2[i] = mySignal1[i];
222                 else
223                         mySignal2[i] = zero;
224
225                 if(i<L)
226                         myFilter2[i] = myFilter1[i];
227                 else
228                         myFilter2[i] = zero;
229         }
230
231         fftwf_execute(forwardPlanSignal);
232         fftwf_execute(forwardPlanFilter);
233
234         for (int i = 0; i < M; i++){
235                 // mySignal2_fft = mySignal2_fft*myFilter2_fft;
236                 float A = mySignal2_fft[i].x;
237                 float B = mySignal2_fft[i].y;
238                 float C = myFilter2_fft[i].x;
239                 float D = myFilter2_fft[i].y;
240                 cufftComplex result;
241                 result.x = A*C-B*D;
242                 result.y = A*D+B*C;
243                 mySignal2_fft[i] = result;
244         }
245
246         fftwf_execute(backwardPlanConv);
247
248         // myConv2 from fftwf must be time reversed and scaled
249         // to match Matlab, myConv1, myConv3, myConv4 and myConv5
250         cufftComplex result;
251         for (int i = 0; i < M; i++){
252                 result.x = myConv2_timeReversed[M-i].x/M;
253                 result.y = myConv2_timeReversed[M-i].y/M;
254                 myConv2[i] = result;
255         }
256         result.x = myConv2_timeReversed[0].x/M;
257         result.y = myConv2_timeReversed[0].y/M;
258         myConv2[0] = result;
259
260         fftwf_destroy_plan(forwardPlanSignal);
261         fftwf_destroy_plan(forwardPlanFilter);
262         fftwf_destroy_plan(backwardPlanConv);
263
264
265         /**
```

```
266         * Time-domain Convolution GPU Using Global Memory
267         */
268        cudaMemcpy(dev_mySignal3, mySignal1, sizeof(cufftComplex)*N, cudaMemcpyHostToDevice);
269        cudaMemcpy(dev_myFilter3, myFilter1, sizeof(cufftComplex)*L, cudaMemcpyHostToDevice);
270
271        int T_B = 512;
272        int B = C/T_B;
273        if(C % T_B > 0)
274                B++;
275        ConvGPU<<<B, T_B>>>(dev_myConv3, dev_mySignal3, dev_myFilter3, N, L);
276
277        cudaMemcpy(myConv3, dev_myConv3, C*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
278
279
280        /**
281         * Time-domain Convolution GPU Using Shared Memory
282         */
283        cudaMemcpy(dev_mySignal4, mySignal1, sizeof(cufftComplex)*N, cudaMemcpyHostToDevice);
284        cudaMemcpy(dev_myFilter4, myFilter1, sizeof(cufftComplex)*L, cudaMemcpyHostToDevice);
285
286        T_B = 512;
287        B = C/T_B;
288        if(C % T_B > 0)
289                B++;
290        ConvGPUshared<<<B, T_B,L*sizeof(cufftComplex)>>>(dev_myConv4, dev_mySignal4,
               dev_myFilter4, N, L);
291
292        cudaMemcpy(myConv4, dev_myConv4, C*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
293
294
295        /**
296         * Frequency-domain Convolution GPU
297         */
298        cufftHandle plan;
299        int n[1] = {M};
300        cufftPlanMany(&plan,1,n,NULL,1,1,NULL,1,1,CUFFT_C2C,1);
301
302        cudaMemset(dev_mySignal5, 0,         M*sizeof(cufftComplex));
303        cudaMemset(dev_myFilter5, 0,         M*sizeof(cufftComplex));
304
305        cudaMemcpy(dev_mySignal5, mySignal2, M*sizeof(cufftComplex), cudaMemcpyHostToDevice);
306        cudaMemcpy(dev_myFilter5, myFilter2, M*sizeof(cufftComplex), cudaMemcpyHostToDevice);
307
308        cufftExecC2C(plan, dev_mySignal5, dev_mySignal5, CUFFT_FORWARD);
309        cufftExecC2C(plan, dev_myFilter5, dev_myFilter5, CUFFT_FORWARD);
310
311        T_B = 512;
312        B = M/T_B;
313        if(M % T_B > 0)
314                B++;
315        PointToPointMultiply<<<B, T_B>>>(dev_mySignal5, dev_myFilter5, M);
316
317        cufftExecC2C(plan, dev_mySignal5, dev_mySignal5, CUFFT_INVERSE);
318
319        T_B = 128;
320        B = M/T_B;
321        if(M % T_B > 0)
322                B++;
323        float scalar = 1.0/((float)M);
324        ScalarMultiply<<<B, T_B>>>(dev_mySignal5, scalar, M);
325
326        cudaMemcpy(myConv5, dev_mySignal5, M*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
327
328        cufftDestroy(plan);
329
330        free(mySignal1);
331        free(mySignal2);
332
```

```
333         free(myFilter1);
334         free(myFilter2);
335 
336         free(myConv1);
337         free(myConv2);
338         free(myConv2_timeReversed);
339         free(myConv3);
340         free(myConv4);
341         free(myConv5);
342         fftwf_cleanup();
343 
344         cudaFree(dev_mySignal3);
345         cudaFree(dev_mySignal4);
346         cudaFree(dev_mySignal5);
347 
348         cudaFree(dev_myFilter3);
349         cudaFree(dev_myFilter4);
350         cudaFree(dev_myFilter5);
351 
352         cudaFree(dev_myConv3);
353         cudaFree(dev_myConv4);
354         cudaFree(dev_myConv5);
355 
356         return 0;
357 }
```

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

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

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

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

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

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

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

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

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

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

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

        __syncthreads();

        cufftComplex temp;
        temp.x = 0;
```

```
64          temp.y = 0;
65          for(int hIdx = 0; hIdx < Lh; hIdx++){
66                  int xAccessIdx = yIdx-hIdx;
67                  if(xAccessIdx>=0 && xAccessIdx<Lx){
68                          // temp += x[xAccessIdx]*h[hIdx];
69                          // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
70                          float A = x[xAccessIdx].x;
71                          float B = x[xAccessIdx].y;
72                          float C = h_shared[hIdx].x;
73                          float D = h_shared[hIdx].y;
74                          cufftComplex complexMult;
75                          complexMult.x = A*C-B*D;
76                          complexMult.y = A*D+B*C;
77
78                          temp.x += complexMult.x;
79                          temp.y += complexMult.y;
80                  }
81          }
82          y[yIdx] = temp;
83  }
84
85  __global__ void PointToPointMultiply(cufftComplex* vec0, cufftComplex* vec1, int maxThreads){
86          int i = blockIdx.x*blockDim.x + threadIdx.x;
87          // Don't access elements out of bounds
88          if(i >= maxThreads)
89                  return;
90          // vec0[i] = vec0[i]*vec1[i];
91          // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
92          float A = vec0[i].x;
93          float B = vec0[i].y;
94          float C = vec1[i].x;
95          float D = vec1[i].y;
96          cufftComplex complexMult;
97          complexMult.x = A*C-B*D;
98          complexMult.y = A*D+B*C;
99          vec0[i] = complexMult;
100 }
101
102 __global__ void ScalarMultiply(cufftComplex* vec0, float scalar, int lastThread){
103         int i = blockIdx.x*blockDim.x + threadIdx.x;
104         // Don't access elements out of bounds
105         if(i >= lastThread)
106                 return;
107         cufftComplex scalarMult;
108         scalarMult.x = vec0[i].x*scalar;
109         scalarMult.y = vec0[i].y*scalar;
110         vec0[i] = scalarMult;
111 }
112
113 int main(){
114         int numBatches = 3104;
115         int N = 12672;
116         int L = 186;
117         int C = N + L - 1;
118         int M = pow(2, ceil(log(C)/log(2)));
119         int maxThreads;
120         int T_B;
121         int B;
122
123         cufftHandle plan;
124         int n[1] = {M};
125         cufftPlanMany(&plan,1,n,NULL,1,1,NULL,1,1,CUFFT_C2C,numBatches);
126
127         // Allocate memory on host
128         cufftComplex *mySignal1;
129         cufftComplex *mySignal1_pad;
130         cufftComplex *myFilter1;
131         cufftComplex *myFilter1_pad;
```

```
132         cufftComplex *myConv1;
133         cufftComplex *myConv2;
134         cufftComplex *myConv3;
135         mySignal1      = (cufftComplex*) malloc(N*numBatches*sizeof(cufftComplex));
136         mySignal1_pad  = (cufftComplex*) malloc(M*numBatches*sizeof(cufftComplex));
137         myFilter1      = (cufftComplex*) malloc(L*numBatches*sizeof(cufftComplex));
138         myFilter1_pad  = (cufftComplex*) malloc(M*numBatches*sizeof(cufftComplex));
139         myConv1        = (cufftComplex*) malloc(C*numBatches*sizeof(cufftComplex));
140         myConv2        = (cufftComplex*) malloc(C*numBatches*sizeof(cufftComplex));
141         myConv3        = (cufftComplex*) malloc(M*numBatches*sizeof(cufftComplex));
142
143         srand(time(0));
144         for(int i = 0; i < N; i++){
145                 mySignal1[i].x = rand()%100-50;
146                 mySignal1[i].y = rand()%100-50;
147         }
148
149         for(int i = 0; i < L; i++){
150                 myFilter1[i].x = rand()%100-50;
151                 myFilter1[i].y = rand()%100-50;
152         }
153
154         cufftComplex zero;
155         zero.x = 0;
156         zero.y = 0;
157         for(int i = 0; i<M*numBatches; i++){
158                 mySignal1_pad[i] = zero;
159                 myFilter1_pad[i] = zero;
160         }
161         for(int batch=0; batch < numBatches; batch++){
162                 for(int i = 0; i < N; i++){
163                         mySignal1[batch*N+i] = mySignal1[i];
164                         mySignal1_pad[batch*M+i] = mySignal1[i];
165                 }
166                 for(int i = 0; i < L; i++){
167                         myFilter1[batch*L+i] = myFilter1[i];
168                         myFilter1_pad[batch*M+i] = myFilter1[i];
169                 }
170         }
171
172         // Allocate memory on device
173         cufftComplex *dev_mySignal1;
174         cufftComplex *dev_mySignal2;
175         cufftComplex *dev_mySignal3;
176         cufftComplex *dev_myFilter1;
177         cufftComplex *dev_myFilter2;
178         cufftComplex *dev_myFilter3;
179         cufftComplex *dev_myConv1;
180         cufftComplex *dev_myConv2;
181         cufftComplex *dev_myConv3;
182         cudaMalloc(&dev_mySignal1, N*numBatches*sizeof(cufftComplex));
183         cudaMalloc(&dev_mySignal2, N*numBatches*sizeof(cufftComplex));
184         cudaMalloc(&dev_mySignal3, M*numBatches*sizeof(cufftComplex));
185         cudaMalloc(&dev_myFilter1, L*numBatches*sizeof(cufftComplex));
186         cudaMalloc(&dev_myFilter2, L*numBatches*sizeof(cufftComplex));
187         cudaMalloc(&dev_myFilter3, M*numBatches*sizeof(cufftComplex));
188         cudaMalloc(&dev_myConv1,   C*numBatches*sizeof(cufftComplex));
189         cudaMalloc(&dev_myConv2,   C*numBatches*sizeof(cufftComplex));
190         cudaMalloc(&dev_myConv3,   M*numBatches*sizeof(cufftComplex));
191
192         /**
193          * Time-domain Convolution GPU Using Global Memory
194          */
195         cudaMemcpy(dev_mySignal1, mySignal1, numBatches*sizeof(cufftComplex)*N,
                cudaMemcpyHostToDevice);
196         cudaMemcpy(dev_myFilter1, myFilter1, numBatches*sizeof(cufftComplex)*L,
                cudaMemcpyHostToDevice);
197
```

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

```
258         cudaFree(dev_mySignal1);
259         cudaFree(dev_mySignal2);
260         cudaFree(dev_mySignal3);
261         cudaFree(dev_myFilter1);
262         cudaFree(dev_myFilter2);
263         cudaFree(dev_myFilter3);
264         cudaFree(dev_myConv1);
265         cudaFree(dev_myConv2);
266         cudaFree(dev_myConv3);
267 
268         return 0;
269 }
```

Chapter 4

Equalizer GPU Implementation

Each equalizer in the PAQ presents an interesting challenge from a GPU implementation perspective. The equations for each equalizer in Section 2.2 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 FFT of the "numerically optimized" detection filter $\mathbf{H}_{\text{NO}}$ and the SOQPSK-TG power spectrum $\mathbf{\Psi}$ are pre-computed and do not require an additional FFT block.

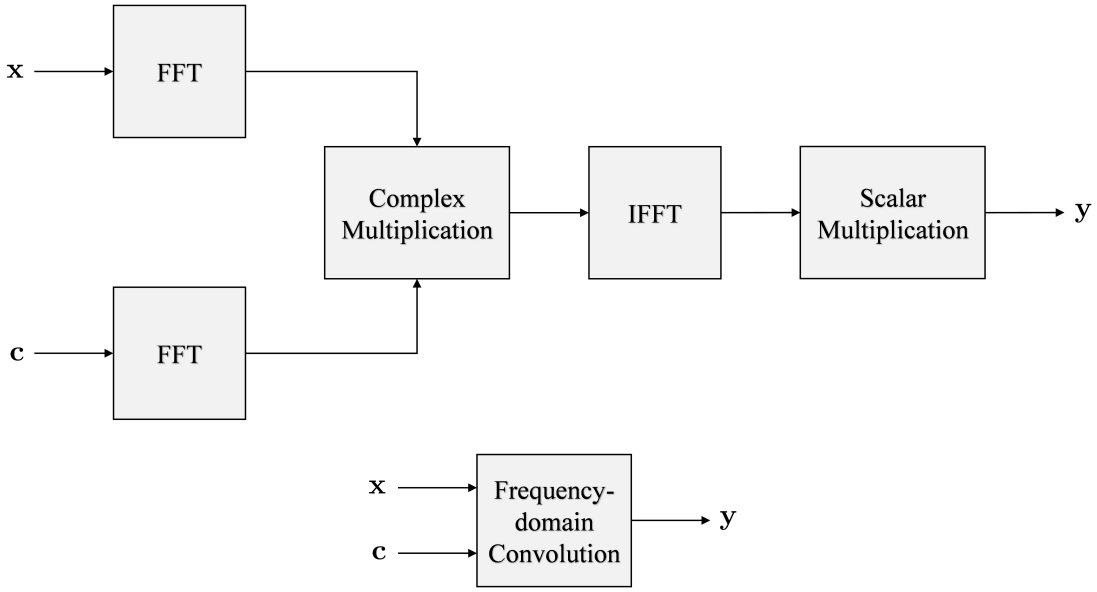


Figure 4.1: To simplify block diagrams, frequency-domain convolution is shown as one block.

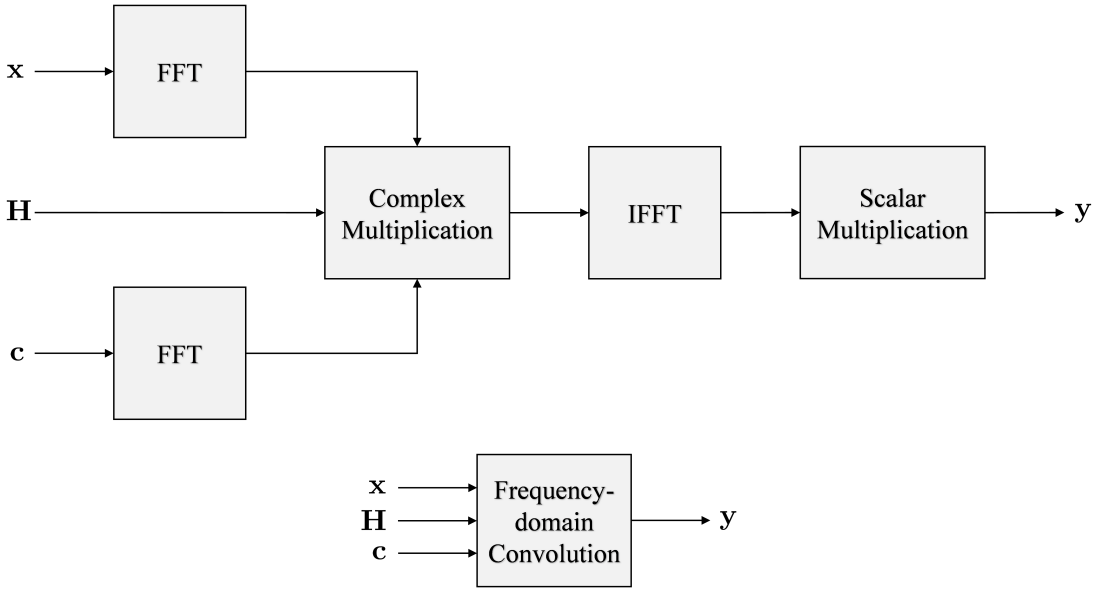


Figure 4.2: To simplify block diagrams, frequency-domain cascaded convolution is shown as one block.

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 (2.19) and (2.30). For reference, the equations are

$$\mathbf{R}_{\hat{h}}\mathbf{c}_{\text{ZF}} = \hat{\mathbf{h}}_{n_0} \quad \text{or} \quad \mathbf{c}_{\text{ZF}} = \mathbf{R}_{\hat{h}}^{-1}\hat{\mathbf{h}}_{n_0} \tag{4.1}$$

$$\mathbf{R}\mathbf{c}_{\text{MMSE}} = \hat{\mathbf{h}}_{n_0} \quad \text{or} \quad \mathbf{c}_{\text{MMSE}} = \mathbf{R}^{-1}\hat{\mathbf{h}}_{n_0} \tag{4.2}$$

where $\mathbf{c}_{\text{ZF}}$, $\mathbf{c}_{\text{MMSE}}$ and $\hat{\mathbf{h}}_{n_0}$ are 186×1 vectors and $\mathbf{R}$ and $\mathbf{R}_{\hat{h}}$ are 186×186 matrices. The only difference between ZF and MMSE is the matrix $\mathbf{R} = \mathbf{R}_{\hat{h}} + 2\hat{\sigma_w^2}\mathbf{I}_{L_1+L_2+1}$.

Before solving Equations (4.1) or (4.1), $\mathbf{R}_{\hat{h}}$, $\mathbf{R}$ and $\hat{\mathbf{h}}_{n_0}$ need to be computed given $\hat{\mathbf{h}}$. The matrices $\mathbf{R}_{\hat{h}}$ and $\mathbf{R}$ require the sample auto-correlation of the estimated channel $\mathbf{r}_{\hat{h}}$. $\hat{\mathbf{h}}_{n_0}$ is the channel estimate time reversed and shifted.

The equalizer filters can be computed by either solving the linear system for $\mathbf{c}_{\text{ZF}}$ and $\mathbf{c}_{\text{MMSE}}$ or by computing the inverse of $\mathbf{R}$ and $\mathbf{R}_{\hat{h}}$ then performing a matrix vector multiplication. 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 are serial. Three approaches to computing the equalizer filter coefficients was explored

- Levinson-Durbin recursion 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.

Levinson-Durbin recursion avoids $\mathcal{O}(n^3)$ operations by using the Toeplitz or diagonal-constant structure of $\mathbf{R}_{\hat{h}}$ and $\mathbf{R}$ [16, Chap. 5]. To begin implementing Levinson-Durbin recursion, a custom GPU kernel was designed for 32-bit *real* floating point data by computing 3104 packets

Table 4.1: Defining start and stop lines for timing comparison in Listing 3.5.

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

of ZF and MMSE equalizer filter coefficients. Levinson-Durbin recursion showed promise by executing on real data in 500 ms. The algorithm was then tested on 32-bit *complex* floating point data by computing 3104 packets of float ZF and MMSE equalizer filter coefficients. Levinson-Durbin recursion was eliminated because execution time for complex filter coefficients was 2500 ms. All processing must be completed in 1907 ms.

The next algorithm explored was computing the inverse of $\mathbf{R}_{\hat{h}}$ and $\mathbf{R}$ using the batch processing cuBLAS library. The cuBLAS library computes a *complex* 32-bit floating point inverse using LU decompositing in 600 ms. cuBLAS executed faster than Levinson-Durbin recursion but 600 ms is still $\%31$ of the total 1907 ms processing time.

The final and fastest algorithm explored solves the linear system using the batch processing cuSolverSp library. "cusolverSpCcsrqrsvBatched" is the GPU function used from the cuSolverSp library. cusolverSpCcsrqrsvBatched is a complex batch solver that leverages the sparse properties of $\mathbf{R}_{\hat{h}}$ and $\mathbf{R}$ by utilizing Compressed Row Storage (CRS) [15]. The Compressed Row Storage reduces the large 186×186 matrices to 12544 element CSR matrices $\mathbf{R}_{\hat{h}\text{CRS}}$ and $\mathbf{R}_{\text{CRS}}$. Before cusolverSpCcsrqrsvBatched 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 [10].

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.

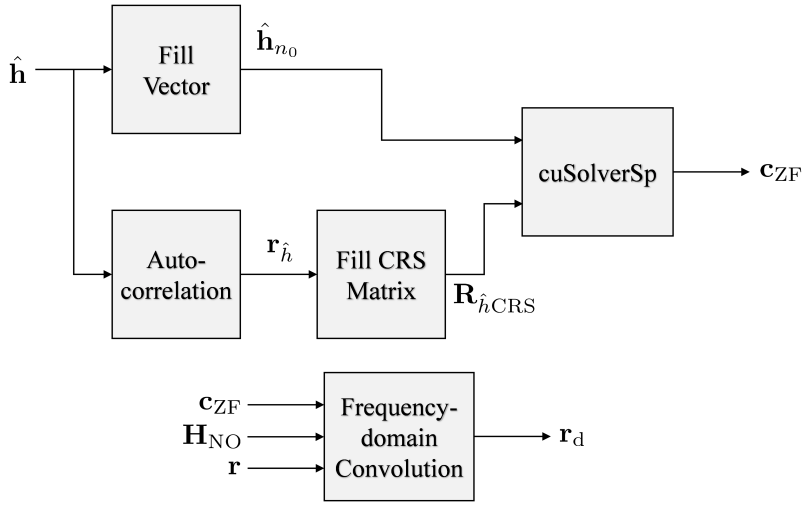


Figure 4.3: Block Diagram showing how the Zero-Forcing equalizer coefficients are implemented in the GPU.

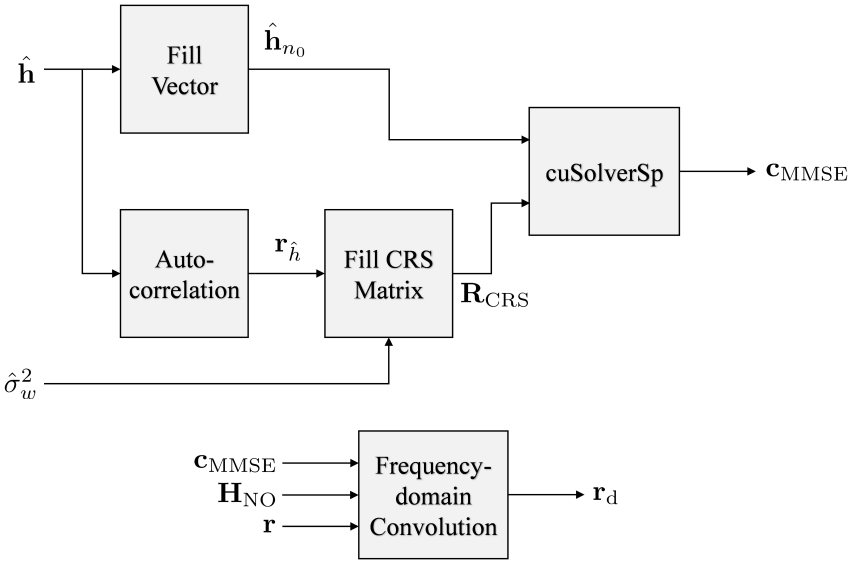


Figure 4.4: Block Diagram showing how the Minimum Mean Squared Error equalizer coefficients are implemented in the GPU.

4.2 Constant Modulus Algorithm GPU Implementation

The Constant Modulus Algorithm (CMA) computes FIR equalizer filter coefficients by a steepest decent algorithm in Equation (2.32). 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 ∇J shown in Equation (2.38). These equations are shown here for reference:

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

$$\nabla J = \frac{1}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} z(n)\mathbf{r}^*(n) \quad or \quad \nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2 \tag{4.4}$$

where

$$z(n) = 2\left[y(n)y^*(n) - 1\right] y(n), \tag{4.5}$$

$$b(n) = \sum_{m=0}^{L_{pkt}-1} z(m)\rho(n-m) \tag{4.6}$$

and

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

The most computationally heavy portion of the CMA equalizer filter implementation is computing the cost function gradient ∇J in Equation (4.4). Section 2.2.2 showed there is two approaches to computing ∇J: directly or using convolution.

The direct approach didn't 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 ∇J directly also applying $\mathbf{c}_{\text{CMA}(b)}$ and computing $\mathbf{c}_{\text{CMA}(b+1)}$.

Using convolution to compute ∇J majorly decreased execution time for the CMA equalizer filter. One CMA iteration took 88.774 ms to execute computing ∇J using frequency-domain

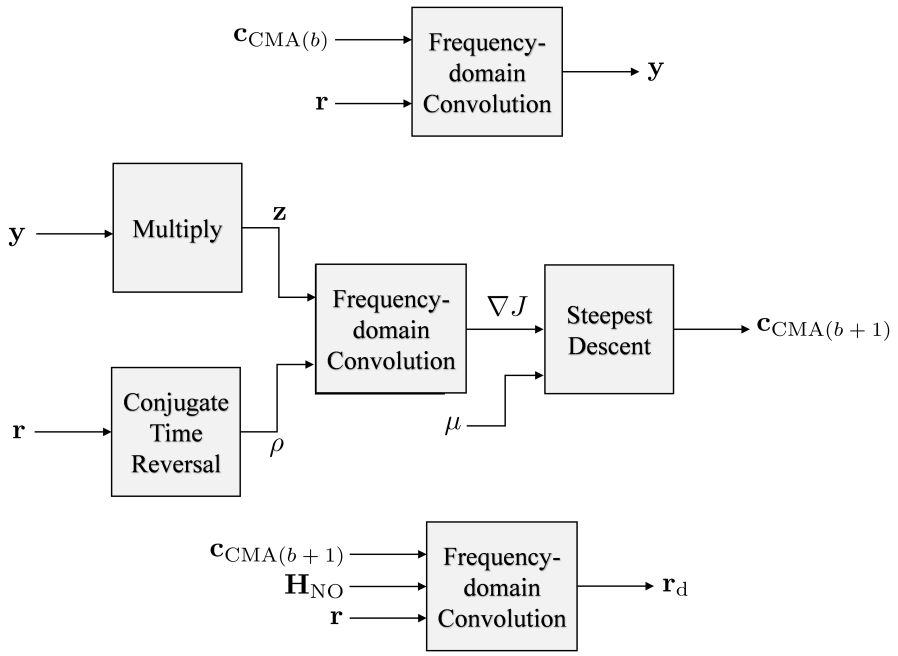


Figure 4.5: Block Diagram showing how the CMA equalizer filter is implemented in the GPU using frequency-domain convolution twice per iteration.

Table 4.2: The gradient vector ∇J can be computed directly or using convolution.

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

convolution also applying $\mathbf{c}_{\mathrm{CMA}(b)}$ and computing $\mathbf{c}_{\mathrm{CMA}(b+1)}$. Note that all other frequency-domain convolution in this thesis 2^{14} or 16384 points, but the convolution length $12672 + 12672 - 1$ is greater than 16384. The FFTs in the computation of $\nabla J(k)$ are 2^{15} or 32768 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 ∇J, the execution time was reduced by 4.74 times. The number of CMA iterations went from 2 iterations using direct implementations to 12 iterations using the convolution implementation.

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

4.3 Frequency Domain Equalizer One and Two GPU Implementation

The Frequency Domain Equalizers (FDEs) were by far the fastest and easiest to implement into GPUs. The block diagram looks just like convolution except that complex multiplication is

$$R_{\text{d1}}(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})H_{\text{NO}}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \cdots, L-1 \tag{4.8}$$

or

$$R_{\text{d2}}(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})H_{\text{NO}}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \cdots, L-1 \tag{4.9}$$

where $R(e^{j\omega_k})$ and $R_{\text{d}}(e^{j\omega_k})$ is the FFT $\mathbf{r}$ and $\mathbf{r}_{\text{d}}$ at ω_k. Equations (4.8) and 4.9 apply FDE1 and FDE2 from Equations (2.45) and (2.46) to the received samples and apply the detection filter. Figures 4.6 and 4.7 show the block diagrams for GPU implementation. As expected, these figures look just like the frequency-domain convolution block diagrams shown in Figures 4.1 and 4.2. Table 4.3 shows the execution times for calculating and applying FDE1 and FDE2.

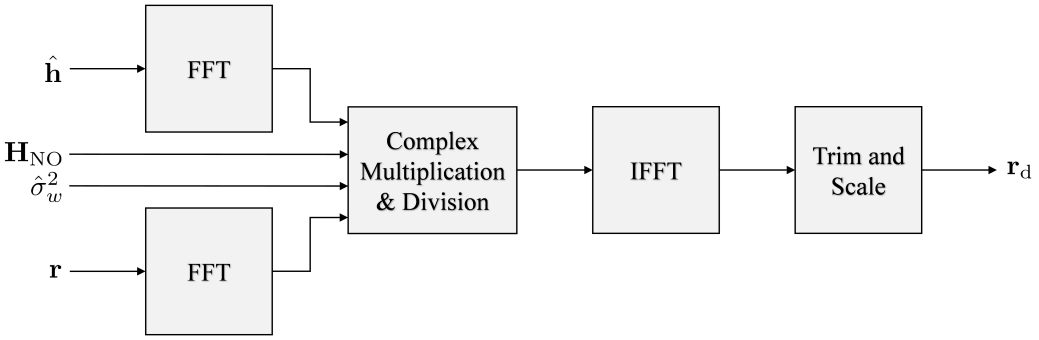


Figure 4.6: Diagram showing Frequency Domain Equalizer One is implemented in the frequency domain in GPUs.

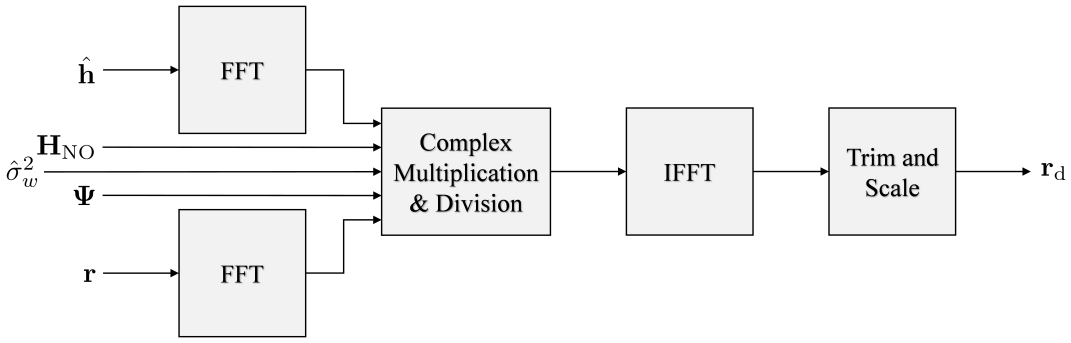


Figure 4.7: Diagram showing Frequency Domain Equalizer Two is implemented in the frequency domain in GPUs.

Chapter 5

Final Summary

this is the final summary

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

[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] M. A. Frerking and P. M. Beauchamp, "Jpl technology readiness level assessment guideline." 3

[5] 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. 9

[6] 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

[7] E. Perrins, "FEC systems for aeronautical telemetry," **IEEE Transactions on Aerospace and Electronic Systems**, vol. 49, no. 4, pp. 2340–2352, October 2013. 13

[8] M. Rice, "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. 15, 17, 19

[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

[10] NVIDIA, "Cuda toolkit documentation," 2017. [Online]. Available: http://docs.nvidia.com/cuda/ 70

[11] Wikipedia, "Graphics processing unit," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Graphics_processing_unit 25

[12] ——, "Fastest fourier transform in the west," 2017. [Online]. Available: http://www.fftw.org/ 42

[13] 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. 42

[14] 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. 48

[15] Wikipedia, "Sparse matrix," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Sparse_matrix 70

[16] M. Hayes, **Statistical Digital Signal Processing and Modeling**. New York: John Wiley & Sons, 1996. 69