# 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**

**2 Problem Statement** **3**

**3 System Overview** **5**

**4 Estimators** **13**

4.1 Preamble Detection . . . . . 14

4.2 Frequency Offset Compensation . . . . . 16

4.3 Channel Estimation . . . . . 16

4.4 Noise Variance Estimation . . . . . 16

4.5 Symbol-by-Symbol Detector . . . . . 17

**5 Signal Processing with GPUs** **19**

5.1 Simple GPU code example . . . . . 19

5.2 GPU kernel using threads and thread blocks . . . . . 22

5.3 GPU Execution and Memory . . . . . 23

5.4 Thread Optimization . . . . . 26

5.5 CPU GPU Pipelining . . . . . 29

**6 GPU Convolution** **35**

6.1 CPU and GPU Single Batch Convolution . . . . . 37

6.2 Batched Convolution . . . . . 44

**7 Equalizer Equations** **63**

7.1 Zero-Forcing and Minimum Mean Square Error Equalizers . . . . . 63

7.1.1 Zero-Forcing . . . . . 64

7.1.2 MMSE Equalizer . . . . . 67

7.2 The Constant Modulus Algorithm . . . . . 68

7.3 The Frequency Domain Equalizers . . . . . 73

7.3.1 Frequency Domain Equalizer One . . . . . 73

7.3.2 Frequency Domain Equalizer Two . . . . . 73

**8 Equalizer GPU Implementation** **75**

8.1 Zero-Forcing and MMSE GPU Implementation . . . . . 75

8.2 Constant Modulus Algorithm GPU Implementation . . . . . 79

8.3 Frequency Domain Equalizer One and Two GPU Implementation . . . . . 81

**9 Final Summary** **83**

**Bibliography** **84**

# List of Tables


5.1 The computational resources available with three NVIDIA GPUs used in this thesis (1x Tesla K40c 2x Tesla K20c). . . . . 26

6.1 Defining start and stop lines for timing comparison in Listing 6.1. . . . . 40

6.2 Convolution computation times with signal length $12672$ and filter length $186$ on a Tesla K40c GPU. . . . . 42

6.3 Convolution computation times with signal length $12672$ and filter length $21$ on a Tesla K40c GPU. . . . . 43

6.4 Defining start and stop lines for timing comparison in Listing 6.2. . . . . 47

6.5 Batched convolution execution times with for a $12672$ sample signal and $186$ tap filter on a Tesla K40c GPU. . . . . 50

6.6 Batched convolution execution times with for a $12672$ sample signal and $21$ tap filter on a Tesla K40c GPU. . . . . 50

6.7 Batched convolution execution times with for a $12672$ sample signal and cascaded $21$ and $186$ tap filter on a Tesla K40c GPU. . . . . 50

6.8 Batched convolution execution times with for a $12672$ sample signal and $206$ tap filter on a Tesla K40c GPU. . . . . 51

7.1 CMA . . . . . 72

8.1 Defining start and stop lines for timing comparison in Listing 6.1. . . . . 78

8.2 The gradient vector $\nabla J(k)$ can be computed using convolution or computed directly. 81

8.3 Defining start and stop lines for timing comparison in Listing 6.1. . . . . 81

## List of Figures


3.1 Functional Block Diagram. . . . . 5

3.2 Computational Flow Block Diagram. . . . . 6

3.3 Processing Block Diagram. . . . . 8

3.4 Picture of full System. . . . . 9

3.5 Picture of Processing MIGHT! . . . . . 10

3.6 Set packet structure. . . . . 11

3.7 The iNET packet structure. . . . . 11

4.1 A block diagram of the estimation process. . . . . 13

4.2 A block diagram of the equalization and symbol detector process. . . . . 14

4.3 Offset Quadriture Phase Shift Keying symbol by symbol detector. . . . . 17

5.1 A block diagram of how a CPU sequentially performs vector addition. . . . . 20

5.2 A block diagram of how a GPU performs vector addition in parallel. . . . . 20

5.3 Block $0$ $32$ threads launched in $4$ thread blocks with $8$ threads per block. . . . . 23

5.4 $36$ threads launched in $5$ thread blocks with $8$ threads per block with $4$ idle threads. 23

5.5 Diagram comparing memory size and speed. Global memory is massive but extremely slow. Registers are extremely fast but there are very few. . . . . 24

5.6 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. . . . . 24

5.7 NVIDIA Tesla K40c and K20c. . . . . 25

5.8 Example of an NVIDIA GPU card. The SRAM is shown to be boxed in yellow. The GPU chip is shown to be boxed in red. . . . . 26

5.9 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. . . . . 29

5.10 Plot showing the number of threads per block doesn't always drastically affect execution time. . . . . 30

5.11 The typical approach of CPU and GPU operations. This block diagram shows the profile of Listing 5.3. . . . . 30

5.12 GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 5.4. . . . . 31

5.13 A block diagram of pipelining a CPU with three GPUs. . . . . 33

6.1 Comparison of number of floating point operations (flops) required to convolve a $12672$ sample complex signal with a varied length tap complex filter. . . . . 37

6.2 Comparison of number of floating point operations (flops) required to convolve a varied length complex signal with a $186$ tap complex filter. . . . . 38

6.3 Comparison of number of floating point operations (flops) required to convolve a varied length complex signal with a $21$ tap complex filter. . . . . 39

6.4 Comparison of a complex convolution on CPU verse GPU. The signal length is varied and the filter is fixed at $186$ taps. The comparison is messy with out lower bounding. . . . . 41

6.5 Comparison of a complex convolution on CPU verse GPU. The signal length is varied and the filter is fixed at $186$ taps. A lower bound was applied by searching for a local minimums in $15$ sample width windows. . . . . 42

6.6 Comparison of a complex convolution on CPU verse GPU. The signal length is varied and the filter is fixed at $21$ taps. A lower bound was applied by searching for a local minimums in $5$ sample width windows. . . . . 43

6.7 Comparison of a complex convolution on CPU verse GPU. The filter length is varied and the signal is fixed at $12672$ samples. A lower bound was applied by searching for a local minimums in $3$ sample width windows. . . . . 44

6.8 Comparison on execution time per batch for complex convolution. The number of batches is varied while the signal and filter length is set to $12672$ and $186$. . . . . 45

6.9 Comparison of a batched complex convolution on a CPU and GPU. The number of batches is varied while the signal and filter length is set to $12672$ and $186$. . . . . 46

6.10 Comparison of a batched complex convolution on a GPU. The signal length is varied and the filter is fixed at $186$ taps. . . . . 47

6.11 Comparison of a batched complex convolution on a GPU. The signal length is varied and the filter is fixed at $21$ taps. . . . . 48

6.12 Comparison of a batched complex convolution on a GPU. The filter length is varied and the signal length is set at $12672$ samples. . . . . 49

6.13 Two ways to convolve the signal $\mathbf{r}$ with the $186$ tap filter $\mathbf{c}$ and $21$ tap filter $\mathbf{d}$. . . . 51

7.1 Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$. . . . . 71

7.2 I need help on this one!!!! . . . . . 74

8.1 Convolution of vectors $\mathbf{c}$ and $\mathbf{r}$ block diagram simplified to one block marked Conv. 76

8.2 Convolution of vectors $\mathbf{c}$, $\mathbf{r}$ and $\mathbf{H}_{\text{NO}}$ block diagram simplified to one block marked Conv. . . . . 76

8.3 Block Diagram showing how the Zero-Forcing equalizer coefficients are implemented in the GPU. . . . . 78

8.4 Block Diagram showing how the Minimum Mean Squared Error equalizer coefficients are implemented in the GPU. . . . . 78

8.5 Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$. . . . . 80

8.6 Diagram showing Frequency Domain Equalizer One is implemented in the frequency domain in GPUs. . . . . 82

8.7 Diagram showing Frequency Domain Equalizer Two is implemented in the frequency domain in GPUs. . . . . 82

# Chapter 1

# Introduction

This is the introduction

# Chapter 2

# Problem Statement

This is the Problem Statement

Some algorithms map very well to CPUs because they are computationally light, but what happens why your CPU cannot acheive the desired throughput or data rate?

In the past, the answer was FPGAs. But now, with Graphics Processing Units getting bigger faster stronger, there has been a recent pull towards GPUs because of

the ease of implementation vs HDL programming

the ease of setup

# Chapter 3

# System Overview

Stuff to Define

$\hat{\mathbf{h}}$

$L_1$

$L_2$

$N_1$

$N_2$

$\mathbf{u}_{n_0}$

$L_{\text{pkt}}$

$L_h$

$p(n)$

Numbers to define

1.907

39321600

12672

3104

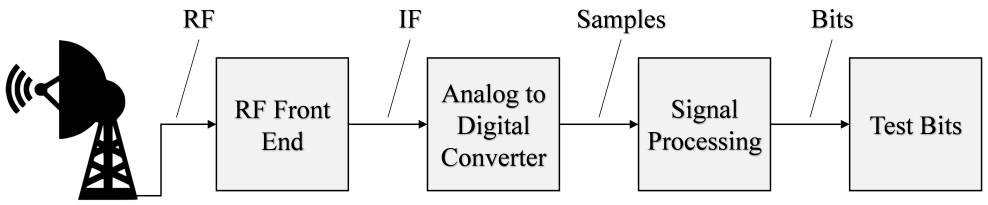



**Figure 3.1:** Functional Block Diagram.

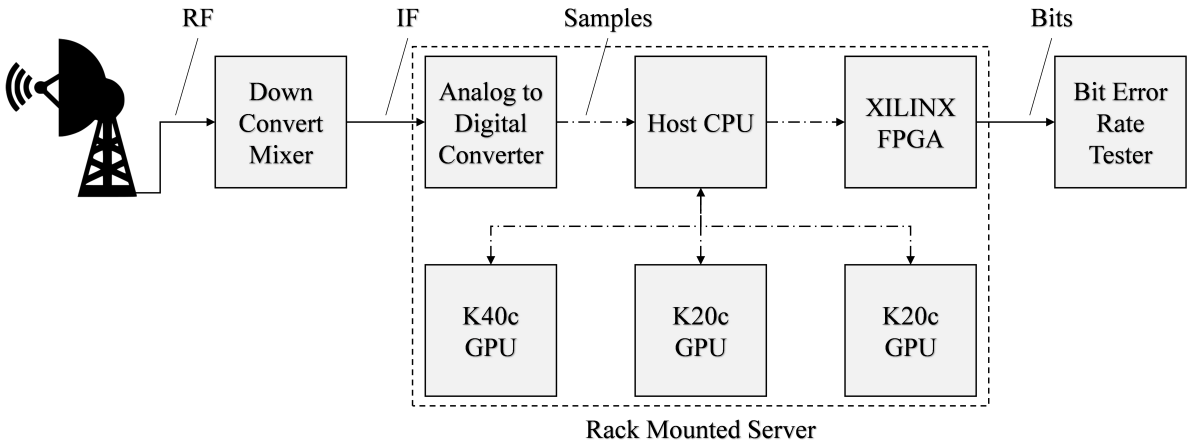



**Figure 3.2:** Computational Flow Block Diagram.

The received samples in this thesis has the iNET packet structure shown in Figure 3.7. The iNET packet consists of a preamble and ASM periodically inserted into the data stream. The iNET preamble and ASM bits are inserted every 6144 data bits. The received signal is sampled at 2 samples/bit, making an $L_{\text{pkt}} = 12672$ sample iNET packet. The iNET preamble comprises eight repetitions of the 16-bit sequence $\text{CD98}_{\text{hex}}$ and the ASM field

$$034776C72728950B0_{\text{hex}}. \tag{3.1}$$

Each 16-bit sequence $\text{CD98}_{\text{hex}}$ sampled at two samples/bit are $L_q = 32$ samples long.

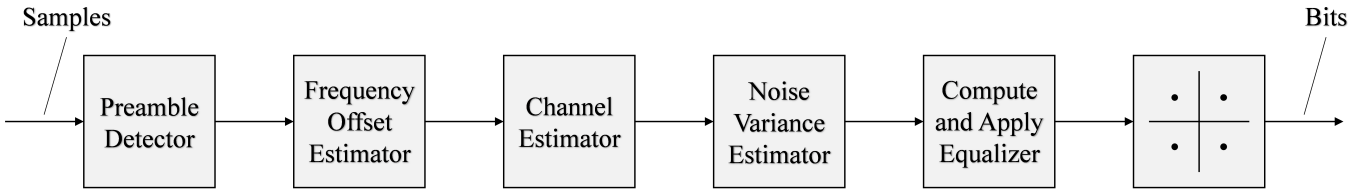



**Figure 3.3:** Processing Block Diagram.


**Figure 3.4:** Picture of full System.

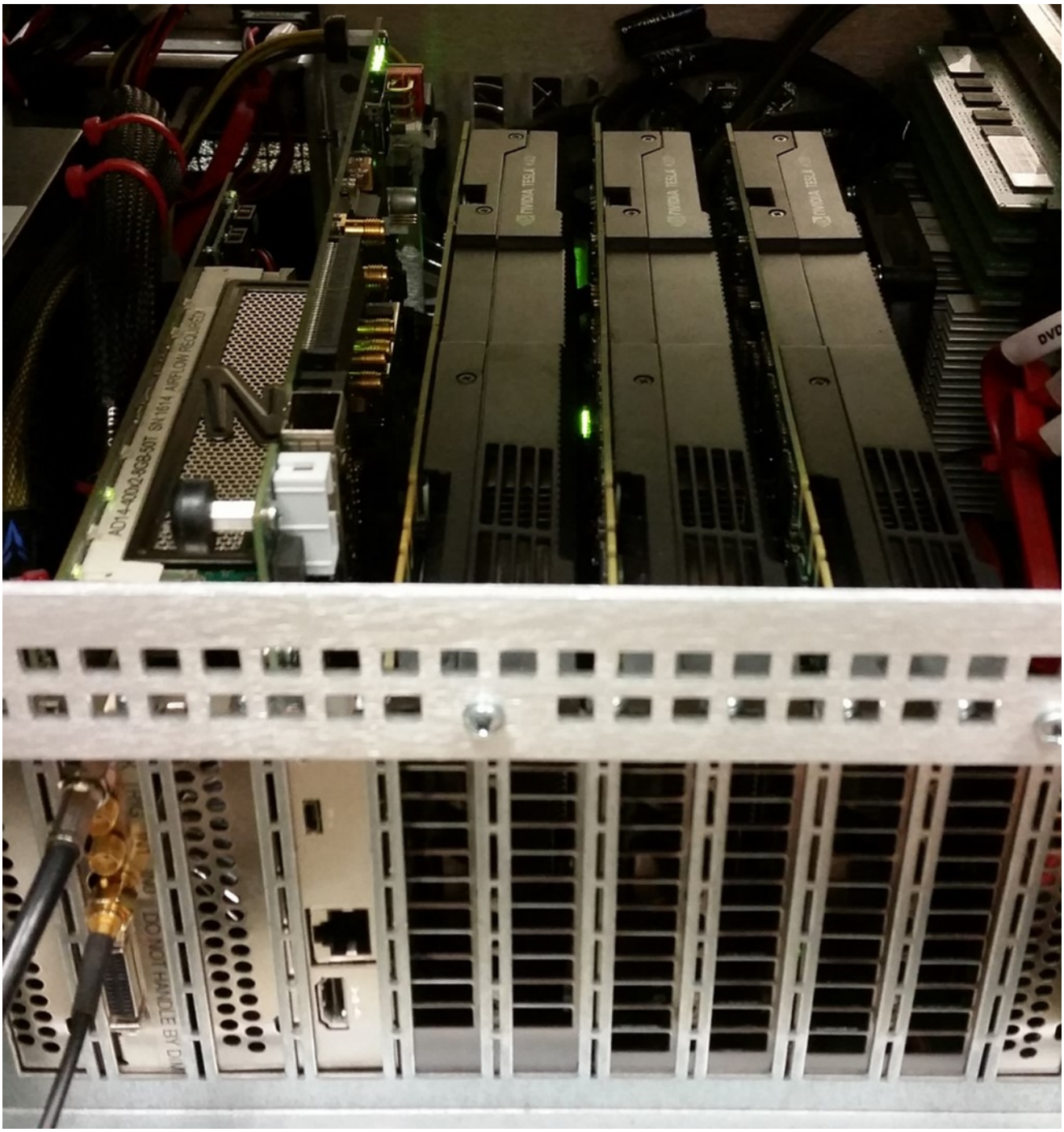



**Figure 3.5:** Picture of Processing MIGHT!

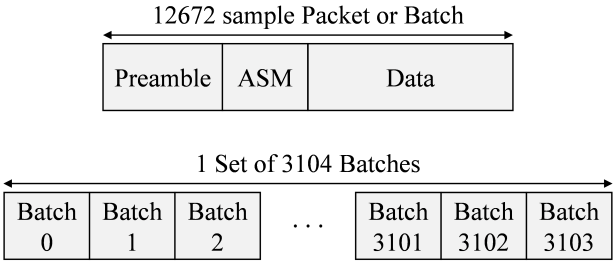



**Figure 3.6:** Set packet structure.

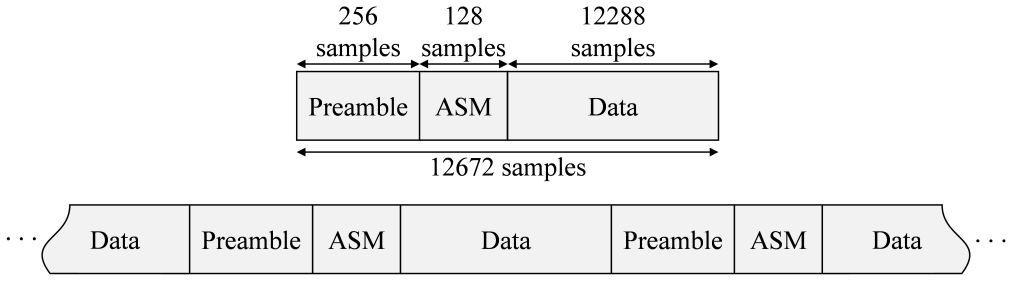



**Figure 3.7:** The iNET packet structure.

# Chapter 4

# Estimators

This thesis studies the GPU implementation of data-aided equalizers. Data-aided equalizers are computed using a channel and noise variance estimates based on a known data sequence in the received signal. The channel and noise variance estimates are very suseptable to a frequency offset. The frequency offset must be estimated then removed. All the estimates are data-aided and thus require finding the known data sequence in the received signal. A preamble detector is employed to estimate the starting index of each preamble in a set. Figure 4.1 shows a block diagram of the estimators in the PAQ project. The estimators will be briefly explained in Sections 4.1 to 4.4

The data-aided equalizers are then computed and the received samples are equalized. With the signal equalized, a detection filter is applied and the symbols are detected by a OQPSK symbol by symbol detector. Figure 4.2 shows a block diagram of how the equalizers are computed then applied. The detection filter and OQPSK detector are explained in Section 4.5 but the blocks in the dotted box. Chapter 7 will explain the equations for the equalizers and the application of the equalizers and detection filter.

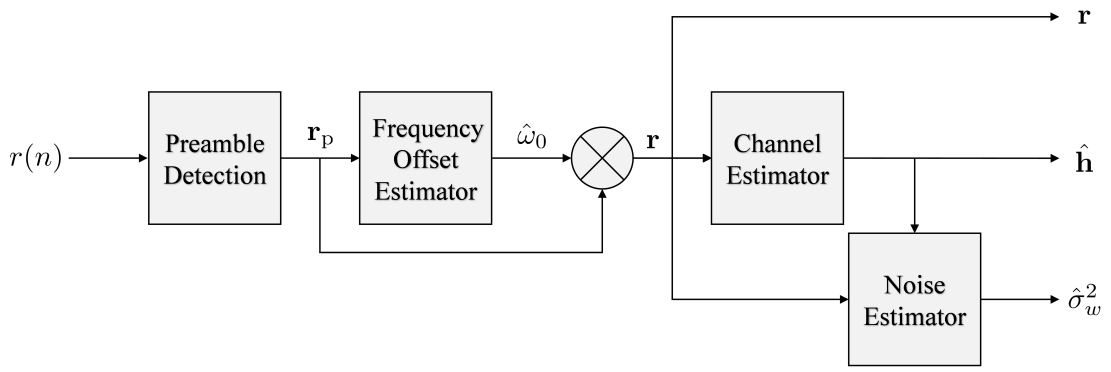



**Figure 4.1:** A block diagram of the estimation process.

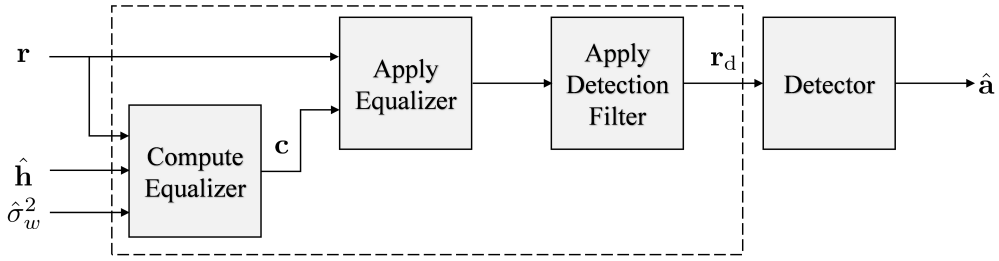



**Figure 4.2:** A block diagram of the equalization and symbol detector process.

## 4.1 Preamble Detection

To compute data-aided preamble assisted equalizers, preambles in the received signal are found then used to estimate various parameters. The goal of the preamble detection step is to structure the received samples into length $L_{\text{pkt}}$ packets or batches with the structure shown in Figure 3.7. Each packet contains $L_{\text{p}} = 256$ preamble samples, $L_{\text{ASM}} = 136$ ASM samples and $L_{\text{d}} = 12288$ data samples. The full length of a packet is $L_{\text{p}} + L_{\text{ASM}} + L_{\text{d}} = 12672$.

Before the structureing the received samples into packets, the preambles are found using a preamble detector explained in [1]. Equations (4.1) through (4.4) have been optimized for GPUs and are implemented directly.

$$L(u) = \sum_{m=0}^{7} \left[ I^2(n,m) + Q^2(n,m) \right] \tag{4.1}$$

where the inner terms are

$$\begin{aligned} I(n,m) \approx \sum_{\ell \in \mathcal{L}_1} r_R(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_2} r_R(\ell+32m+n) + \sum_{\ell \in \mathcal{L}_3} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_4} r_I(\ell+32m+n) \\ + 0.7071 \left[ \sum_{\ell \in \mathcal{L}_5} r_R(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_6} r_R(\ell+32m+n) \right. \\ \left. + \sum_{\ell \in \mathcal{L}_7} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_8} r_I(\ell+32m+n) \right], \end{aligned} \tag{4.2}$$

and

$$
\begin{aligned}
Q(n,m) \approx & \sum_{\ell \in \mathcal{L}_1} r_I(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_2} r_I(\ell + 32m + n) \\
& - \sum_{\ell \in \mathcal{L}_3} r_R(\ell + 32m + n) + \sum_{\ell \in \mathcal{L}_4} r_R(\ell + 32m + n) \\
& + 0.7071 \left[ \sum_{\ell \in \mathcal{L}_5} r_I(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_6} r_I(\ell + 32m + n) \right. \\
& \left. - \sum_{\ell \in \mathcal{L}_7} r_R(\ell + 32m + n) + \sum_{\ell \in \mathcal{L}_8} r_R(\ell + 32m + n) \right] \qquad (4.3)
\end{aligned}
$$

with

$$
\begin{aligned}
\mathcal{L}_1 &= \{0, 8, 16, 24\} \\
\mathcal{L}_2 &= \{4, 20\} \\
\mathcal{L}_3 &= \{2, 10, 14, 22\} \\
\mathcal{L}_4 &= \{6, 18, 26, 30\} \\
\mathcal{L}_5 &= \{1, 7, 9, 15, 17, 23, 25, 31\} \\
\mathcal{L}_6 &= \{3, 5, 11, 12, 13, 19, 21, 27, 28, 29\} \\
\mathcal{L}_7 &= \{1, 3, 9, 11, 12, 13, 15, 21, 23\} \\
\mathcal{L}_8 &= \{5, 7, 17, 19, 25, 27, 28, 29, 31\}.
\end{aligned} \qquad (4.4)
$$

A correlation peak in $L(u)$ indicate the starting index $k$ of a preamble. The vector $\mathbf{r}_{\text{p}}$ in Figure 4.1 is defined by

$$
\mathbf{r}_{\text{p}} = \begin{bmatrix} r(k) \\ \vdots \\ r(k + L_{\text{pkt}} - 1) \end{bmatrix} = \begin{bmatrix} r_{\text{p}}(0) \\ \vdots \\ r_{\text{p}}(L_{\text{pkt}} - 1) \end{bmatrix} \qquad (4.5)
$$

## 4.2 Frequency Offset Compensation

The frequency offset estimator shown in Figure 4.1 is the estimator taken from [2, eq. (24)]. With the notation adjusted slightly, the frequency offset estimate is

$$\hat{\omega}_0 = \frac{1}{L_q} \arg \left\{ \sum_{n=i+2L_q}^{i+7L_q-1} r_{\mathrm{p}}(n) r_{\mathrm{p}}^*(n - L_q) \right\} \quad \text{for } i = 1, 2, 3, 4, 5. \tag{4.6}$$

The frequency offset is estimated for every packet or each vector $\mathbf{r}_{\mathrm{p}}$. The frequency offset is compensated for by derotating the packet structured samples by the estimated offset

$$r(n) = r_{\mathrm{p}}(n) e^{-j\hat{\omega}_0 n}. \tag{4.7}$$

Equations (4.6) and (4.7) are easily implemented into GPUs.

## 4.3 Channel Estimation

The channel estimator is the ML estimator taken from [3, eq. 8].

$$\hat{\mathbf{h}} = \underbrace{\left(\mathbf{X}^\dagger \mathbf{X}\right)^{-1} \mathbf{X}^\dagger}_{\mathbf{X}_{\mathrm{lpi}}} \mathbf{r} \tag{4.8}$$

where $\mathbf{X}$ is a convolution matrix formed from the ideal preamble and ASM samples and $\mathbf{X}_{\mathrm{lpi}}$ is the left psudo-inverse of $\mathbf{X}$. The ML channel estimator is the result of the matrix operation

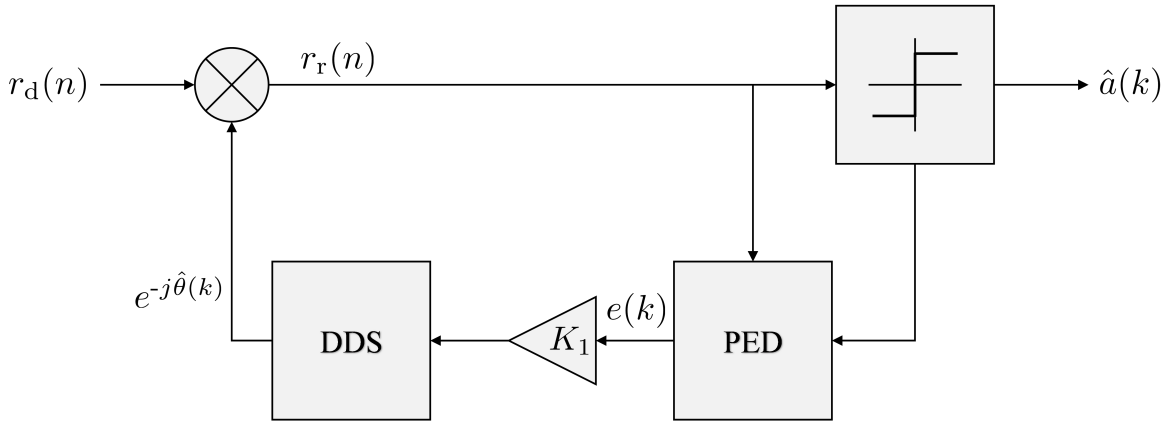
$$\hat{\mathbf{h}} = \mathbf{X}_{\mathrm{lpi}} \mathbf{r}. \tag{4.9}$$

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

## 4.4 Noise Variance Estimation

The noise variance estimator is also taken from [3, eq. 9]

$$\hat{\sigma}_w^2 = \frac{1}{2\rho} \left| \mathbf{r} - \mathbf{X}\hat{\mathbf{h}} \right|^2 \tag{4.10}$$




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

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

**Figure 4.3:** Offset Quadriture Phase Shift Keying symbol by symbol detector.

where

$$\rho = \mathrm{Trace}\left\{\mathbf{I} - \mathbf{X}\left(\mathbf{X}^{\dagger}\mathbf{X}\right)^{-1}\mathbf{X}^{\dagger}\right\}. \tag{4.11}$$

Equation (4.10) is easily implemented into GPUs.

### 4.5 Symbol-by-Symbol Detector

Before the symbols are detected, a detection filter is applied then the signal is downsampled by 2. The detection filter is a "numerically optimized" SOQPSK detection filter $H_{\mathrm{NO}}$ [4, Fig. 3]. The symbol by symbol detector block in Figure 4.2 is a Offset Quadrature Phase Shift Keying (OQPSK) detector. Using the simple OQPSK detector in place of a complex MLSE SOQPSK-TG detector leads to less than 1dB in bit error rate [4].

A Phase Lock Loop (PLL) is needed in the OQPSK detector to track out residual frequency offset. The residual frequency offset results from the 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. The detector and PLL are parallelized 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.

# Chapter 5

# Signal Processing with GPUs

This thesis explores the use of GPUs in data-aided estimation, equalization and filtering operations. The purpose of chapter is to provide context for the contributions of this thesis. As such this overview is not a tutorial. For a full explination of CUDA programming please see the CUDA toolkit documentation [6].

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. Recently, general purpose GPUs are being used for high performance computing in computer vision, deep learning, artificial intelligence and signal processing [5].

GPUs cannot be programmed the way as a CPU. NVIDIA released a extension to C, C++ and Fortran called CUDA (Compute Unified Device Architecture). CUDA allows a programmer to write C++ like functions that are massively parallel called *kernels*. To invoke parallelism, a GPU kernel is called $N$ times and mapped to $N$ *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.

## 5.1 Simple GPU code 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 where the memory physically is 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, the parallel GPU kernels can be called. After the GPU kernels have finished, the results have to be copied back from the device (GPU) to the host (CPU).

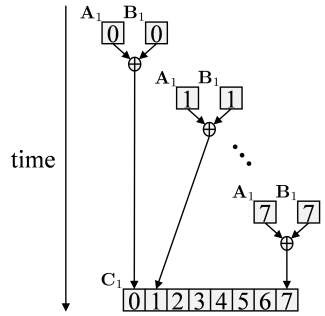



**Figure 5.1:** A block diagram of how a CPU sequentially performs vector addition.

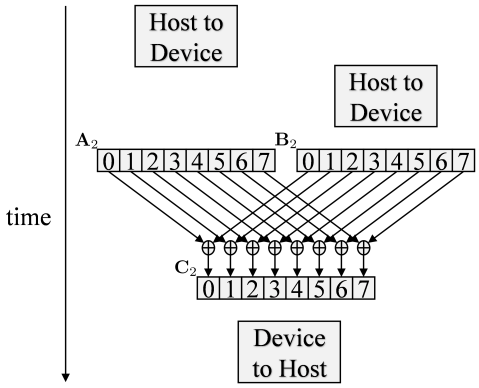



**Figure 5.2:** A block diagram of how a GPU performs vector addition in parallel.

Listing 5.1 shows a simple program that sums two vectors together where each vector is length $1024$.

$$\begin{aligned} \mathbf{C}_1 &= \mathbf{A}_1 + \mathbf{B}_1 \\ \mathbf{C}_2 &= \mathbf{A}_2 + \mathbf{B}_2 \end{aligned} \tag{5.1}$$

On line $42$ the CPU computes $\mathbf{C}_1$ by summing elements of $\mathbf{A}_1$ and $\mathbf{B}_1$ together *sequentially*. Figure 5.1 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 5.2 shows how the GPU computes $\mathbf{C}_2$ *in parallel*.

**Listing 5.1:** 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 numPoints = pow(2,22);
        cout << numPoints << endl;
        /**
        * Vector Addition on CPU
        */
        // allocate memory on host
        float *A1;
        float *B1;
        float *C1;
        A1 = (float*) malloc (numPoints*sizeof(float));
        B1 = (float*) malloc (numPoints*sizeof(float));
        C1 = (float*) malloc (numPoints*sizeof(float));

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

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

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

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

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

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

        // Compute number of blocks for set number of threads
```

```
67          int numBlocks = numPoints/numTreadsPerBlock;
68
69          // If there are left over points, run an extra block
70          if(numPoints % numTreadsPerBlock > 0)
71                  numBlocks++;
72
73          // Run computation on device
74          //for(int i = 0; i < 100; i++)
75          VecAddGPU<<<numBlocks, numTreadsPerBlock>>>(C2_gpu, A2_gpu, B2_gpu, numPoints);
76
77          // Copy vector C2 from device to host
78          cudaMemcpy(C2, C2_gpu, sizeof(float)*numPoints, cudaMemcpyDeviceToHost);
79
80          // Compare C2 to C1
81          bool equal = true;
82          for(int i = 0; i < numPoints; 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 }
```

### 5.2 GPU kernel using threads and thread blocks

A GPU kernel is executed on a GPU by launching numBlocks thread blocks with a set number of threads per block. In the Listing 5.1, VecAddGPU is launched with 32 threads per block on line 75. 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 atleast $2^22$ threads or $131072 = 2^22/32$ 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. blockDim is the number of threads assigned per block, in fact blockDim = numTreadsPerBlock. 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.

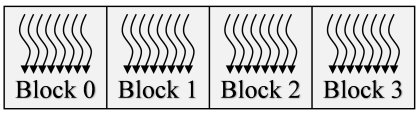



**Figure 5.3:** Block $0$ $32$ threads launched in $4$ thread blocks with $8$ threads per block.

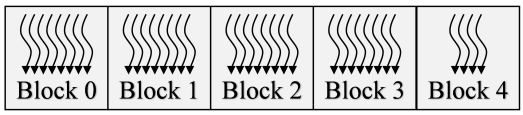



**Figure 5.4:** $36$ threads launched in $5$ thread blocks with $8$ threads per block with $4$ idle threads.

To turn a CPU for loop in to a GPU kernel that runs $0$ to $N-1$, the GPU kernel will launch atleast $N$ threads are with $T$ threads per thread block. The number of blocks need is $M = \frac{N}{T}$ or $M = \frac{N}{T} + 1$ if $N$ is not an integer multiple of $T$. Figure 5.3 shows $32$ threads launched in $4$ thread blocks with $8$ threads per block. Figure 5.4 shows $36$ threads launched in $5$ thread blocks with $8$ threads per block. An full extra thread block is launched with $8$ threads but $4$ threads are idle.

## 5.3 GPU Execution and Memory

Thread blocks are executed independent of other thread blocks. The GPU does not guarantee Block $0$ will execute before Block $2$. Threads in individual blocks can coordinate and use shared memory but blocks do not coordinate with other blocks. Threads have access to private local memory that is fast and efficient. Each thread in a thread block has access to shared memory the is private to the thread block. All threads have access to global memory.

Local memory is the fastest and global memory is by far the slowest. One global memory access takes 400-800 clock cycles while a local memory in the form of registers, L1 and shared memory is a few clock cycles. Figure 5.5 helps visualize the trade offs of memory and Figure 5.6 shows where each type of memory is located.

Why not just use local memory for all computation storage? Elements need to come from global memory to before they can be used in local memory. If many threads access the same elements in global memory, clock cycles can be saved by copying the elements from global to

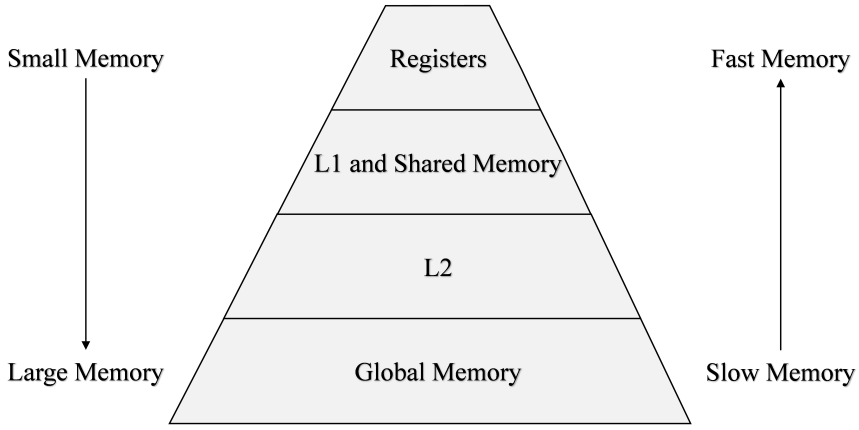



**Figure 5.5:** Diagram comparing memory size and speed. Global memory is massive but extremely slow. Registers are extremely fast but there are very few.

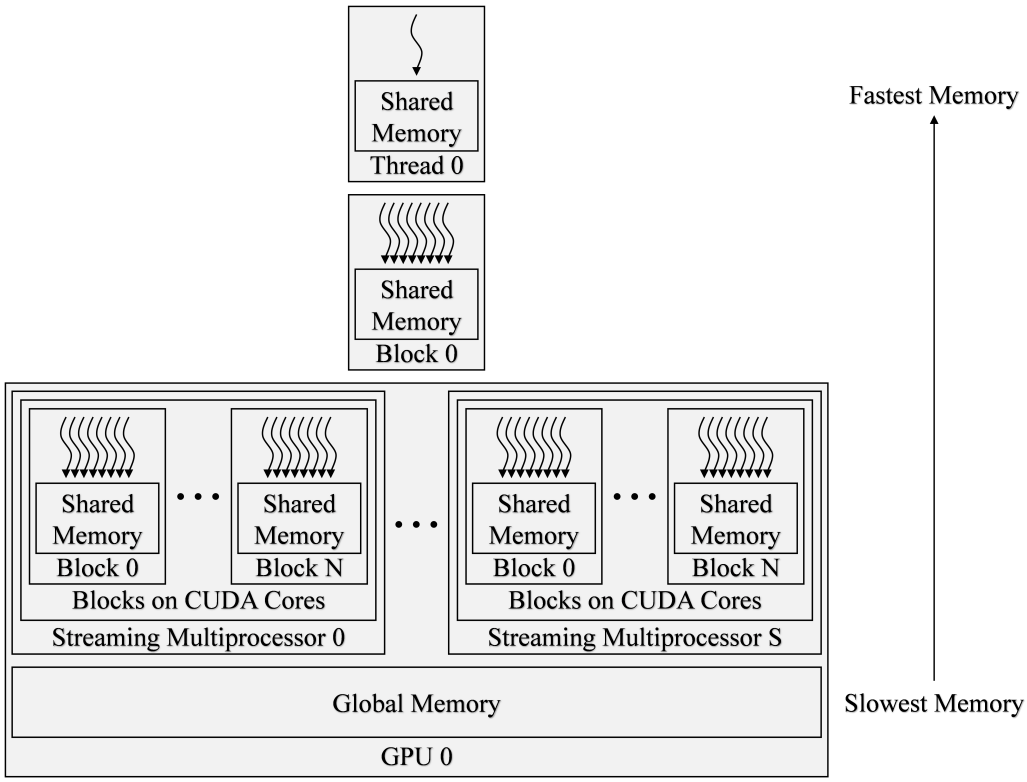



**Figure 5.6:** A block diagram where local, shared, and global memory is located. Each thread has private local memory. Each thread block has private shared memory. The GPU has global memory that all threads can access.


**Figure 5.7:** NVIDIA Tesla K40c and K20c.

shared memory. Local and shared memory should be used as much as possible but sometimes a GPU kernel cant utilized local and shared memory because elements might only be used once.

Why is global memory so slow? Looking at the physical hardware is instructive. This thesis uses NVIDIA Tesla K40c and K20c GPUs, Table 5.1 lists some specifications and Figure 5.7 shows the form factor of the these GPUs. The red box in Figure 5.8 shows the GPU chip and the yellow boxes show the SRAM that is *off* the GPU chip.

The global memory is located in the SRAM. To move memory to thread blocks *on* the GPU chip from global memory requires fetching memory from *off* the GPU. Considering that global memory is off chip, 400-800 clock cycles doesn't sound all that bad.

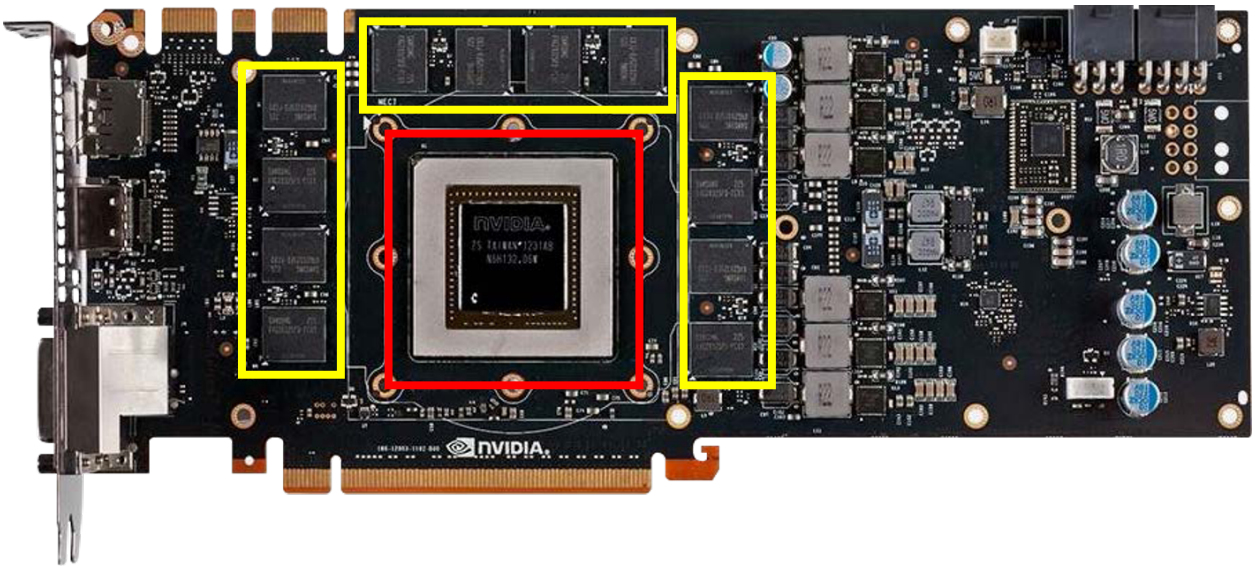


**Figure 5.8:** Example of an NVIDIA GPU card. The SRAM is shown to be boxed in yellow. The GPU chip is shown to be boxed in red.

| Feature | Tesla K40c | Tesla K20c |
|---|---|---|
| Memory size (GDDR5) | 12 GB | 5 GB |
| CUDA cores | 2880 | 2496 |
| Base clock (MHz) | 745 | 732 |

**Table 5.1:** The computational resources available with three NVIDIA GPUs used in this thesis (1x Tesla K40c 2x Tesla K20c).

## 5.4 Thread Optimization

When writing a custom GPU kernel, it is tempting to launch as many threads per block as possible. Launching 256 threads per block doesn't sound as fast at launching 1024 threads per block, right? Wrong. Running the GPU at low occupancy provides each thread with more resources in the each block. Running the GPU at high occupancy provides each thread with less resources.

Launching 1024 threads per block isnt always bad though. 1024 might be the optimal number of threads per block. If a GPU kernel was very computationally heavy but didn't require much memory resources, 1024 threads might be a good amount of threads per block.

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, $1$ to $1024$. Listing 5.2 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 5.2:** Code snippet for thread optimization.

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

        GPUkernel<<<numBlocks, numTreadsPerBlock>>>(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;
                numTreadsPerBlock_opt = numTreadsPerBlock;
        }
}
cout << "Optimal Threads Per Block " << numTreadsPerBlock_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 resources.

Figure 5.9 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, this example kernel can have a $2\times$ speed up.

Adjusting the number of threads per block doesn't always drastically speed up GPU kernels. Figure 5.9 shows the execution time for another GPU kernel with varying threads per block. Launching $560$ does produce about a $1.12\times$ speed up.

While writing a custom GPU kernel then figuring out how to optimize it is extremely satisfying, CUDA has super optimized GPU libraries that are extremely useful and efficient. The CUDA libraries are written by NVIDIA engineers that know how to squeeze out every drop of performance out of NVIDIA GPUs. Some libraries used in this thesis are cuFFT, cuBLAS and cuSolverSp.

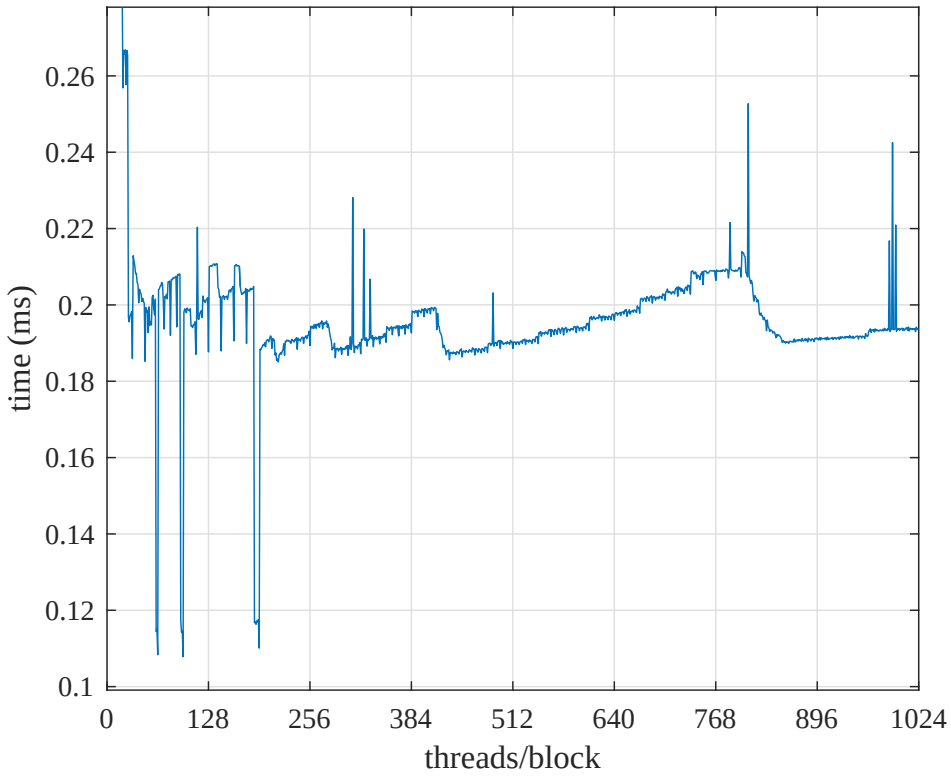



**Figure 5.9:** Plot showing how execution time is affected by changing the number of threads per block. The optimal execution time for an example GPU kernel is 0.1078ms at the optimal 96 threads per block.

## 5.5 CPU GPU Pipelining

A basic program flow is shown in Listing 5.3. The CPU acquires data from myADC on Line 5. After taking time to acquire data, the data is copied to the CPU, the data is processed in the GPU then result is copied back to the CPU on Lines 8 to 10. cudaDeviceSynchronize on line 13 causes the CPU to wait until all instructions on the GPU are complete. Acquiring and copying data takes precious processing time. What if the GPU could be processing data while the CPU acquires and copied data? How much computation time could be gained by pipelining acquiring data and processing data? How much would the throughput increase?

Figure 5.11 shows a block diagram of what is happening on the CPU and GPU in Listing 5.3. The GPU is idle while the CPU is acquiring data. The CPU is idle while the GPU is processing and data is being transferred to and from the GPU.

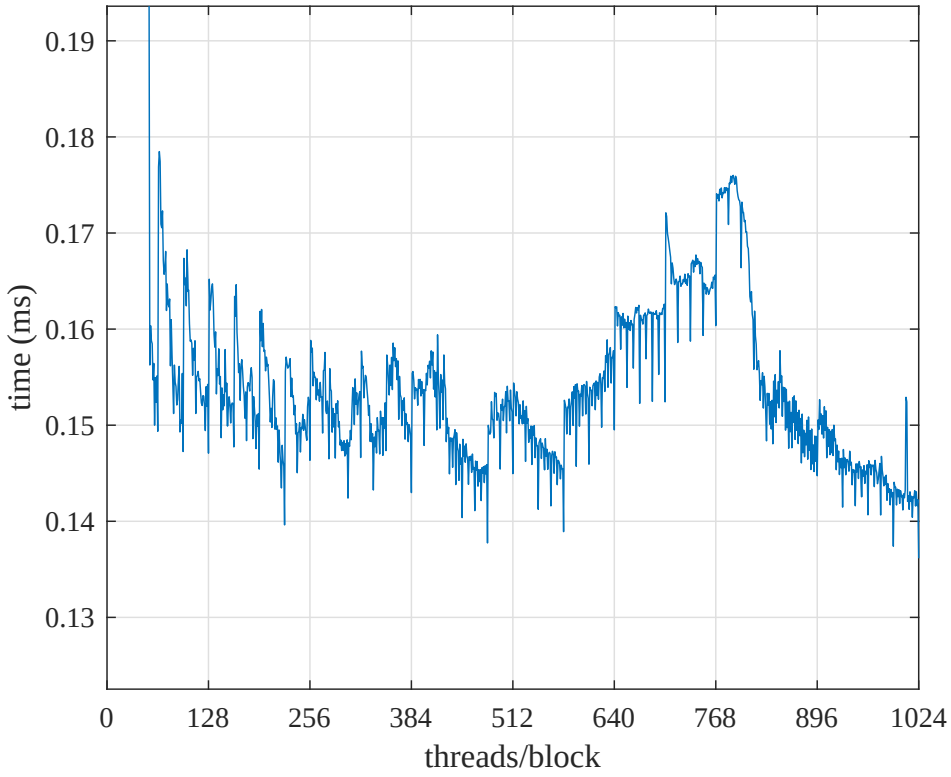



**Figure 5.10:** Plot showing the number of threads per block doesn't always drastically affect execution time.

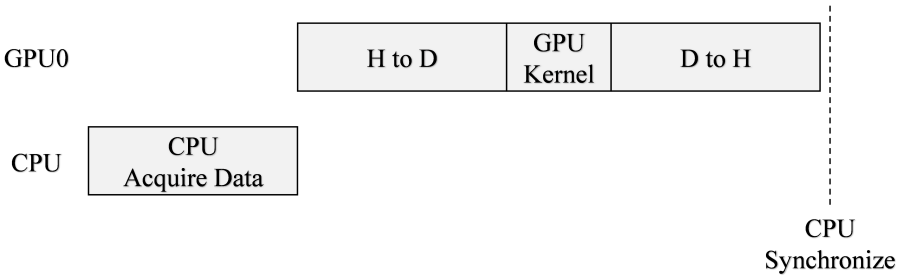



**Figure 5.11:** The typical approach of CPU and GPU operations. This block diagram shows the profile of Listing 5.3.

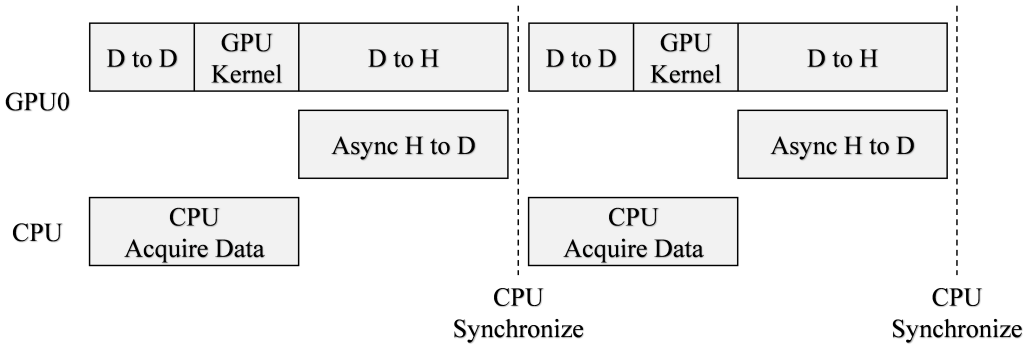



**Figure 5.12:** GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 5.4.

**Listing 5.3:** Example code Simple example of the CPU acquiring data from myADC, copying from host to device, processing data on the device then copying from device to host. No processing occurs on device while CPU is acquiring data.

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

Can the throughput increase by using idle time on the GPU and CPU? Yes, CPU and GPU operations can sacrifice latency for throughput by pipelineing. After the CPU gives instructions to the GPU, the CPU can do other operations like acquire data or perform algorithms better suited for CPUs than the GPUs. Once the CPU has finished its operations, the CPU can wait for the GPU to finish.

Listing 5.4 shows how to pipeline CPU and GPU operations. Assuming data is already on the GPU from a prior iteration, the CPU gives instructions to the GPU then starts acquiring 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. This system suffers a full cycle latency.

**Listing 5.4:** 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();
        ...
}
```

Pipelineing can be extended to multiple GPUs for even more throughput but only suffer latency of copying memory to one GPU. Figure 5.13 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.

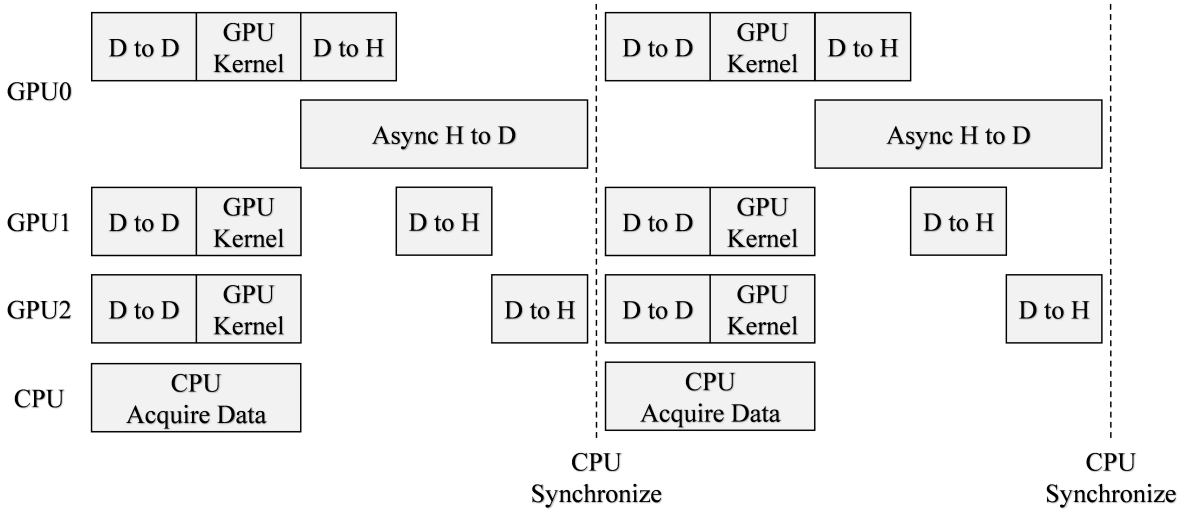



**Figure 5.13:** A block diagram of pipelining a CPU with three GPUs.

# Chapter 6

# GPU Convolution

Convolution is one of the most important tools in digital signal processing. The PAQ system explained in Chapter 3 uses convolution at least 10 times, depending on the number of CMA iterations. If convolution execution time improves by 10 ms, the full system execution time improves by 100 ms. This chapter explores the how to optimize GPU convolution.

Discrete time convolution can be implemented in the time or frequency domain. Discrete time convolution computed in the time domain is

$$y(n) = \sum_{m=0}^{L-1} x(m)h(n-m) \tag{6.1}$$

and discrete time convolution computed in the frequency domain is

$$\mathbf{y} = \mathscr{F}^{-1}(\mathscr{F}(\mathbf{x}) \times \mathscr{F}(\mathbf{h})) \tag{6.2}$$

where the $N$ sample complex signal $\mathbf{x}$ is convolved with the $L$ tap filter complex $\mathbf{h}$.

Traditionally the number of flops is used to estimate how computationally intense an algorithm is. Each complex multiply

$$(A + jB) \times (C + jD) = (AC - BD) + j(AD + BC) \tag{6.3}$$

is $6$ flops, $4$ multiplies and $2$ additions/subtractions. Each output element of $\mathbf{y}$ in Equation (6.1) requires $8L$ flops. Each term in the $L$ long summation takes $8$ flops, $6$ flops per multiply plus $2$ flops (real and imaginary) for the sum. The output vector $\mathbf{y}$ is $N+L-1$ samples long. The number

of flops required for convolution is

$$8L(N + L - 1) \text{ flops.} \tag{6.4}$$

The length of the convolution, $M = N + L - 1$ is the minimum point Fourier Transform possible. To leverage the Cooley-Tukey radix 2 Fast Fourier Transform (FFT), it is common practice append zeros to the next power of to above $M$. The current most popular CPU based FFT is the Fastest Fourier Transform in the West (FFTW) library, FFTW uses the Cooley-Tukey radix 2 transform. Each radix 2 forward or backward Fourier transform requires $5M \log_2(M)$ flops [8, 9]. As shown by Equation 6.2, frequency domain convolution requires

$$3 \times 5M \log_2(M) + 6M \text{ flops} \tag{6.5}$$

from $3$ FFTs and a length $M$ point to point multiply.

Comparing Equations 6.4 and 6.5, if a signal or filter length is relatively long the frequency domain is the best choice. What constitutes a "long" signal or filter? When should convolution be done in the frequency domain rather than time domain?

Figure 6.1 compares the number of flop required to convolve a $12672$ sample complex signal with a varied length tap complex filter. According to the number of flops in the figure, frequency domain convolution requires less flops if the filter is longer $40$ taps.

Figure 6.2 compares the number of flops required for time domain verse frequency domain convolution of a $12672$ sample complex signal with a $186$ tap complex filter. Figure 6.3 compares the number of flops required for time domain verse frequency domain convolution of a $12672$ sample complex signal with a $21$ tap complex filter. Appending zeros to the next power of $2$ causes the stair stepping pattern. Judging by the figures with varied signal lengths, a $186$ tap filter is "long" and a $21$ tap filter is "short."

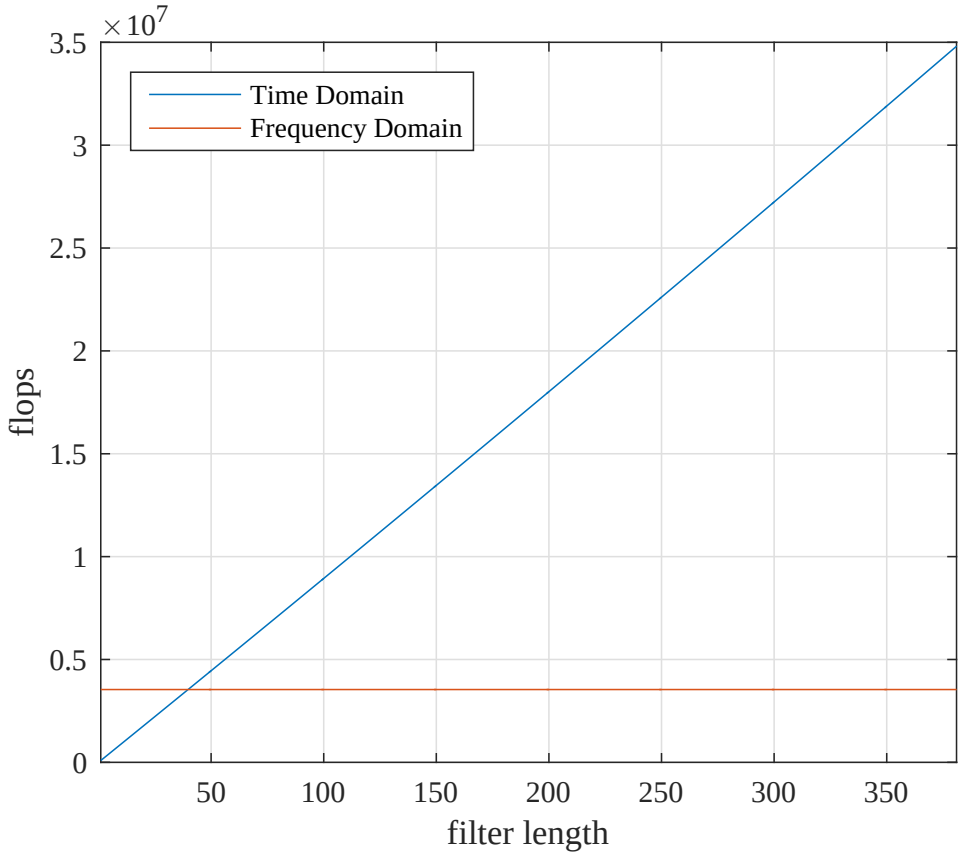



**Figure 6.1:** Comparison of number of floating point operations (flops) required to convolve a 12672 sample complex signal with a varied length tap complex filter.

## 6.1 CPU and GPU Single Batch Convolution

With an understanding of the number of flops in time verses frequency domain required to implement convolution, do the number of flops have a direct relationship to execution time in CPUs and GPUS? To explore the flop to execution time relationship Listing 6.1 shows five different ways of implementing convolution:

- 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

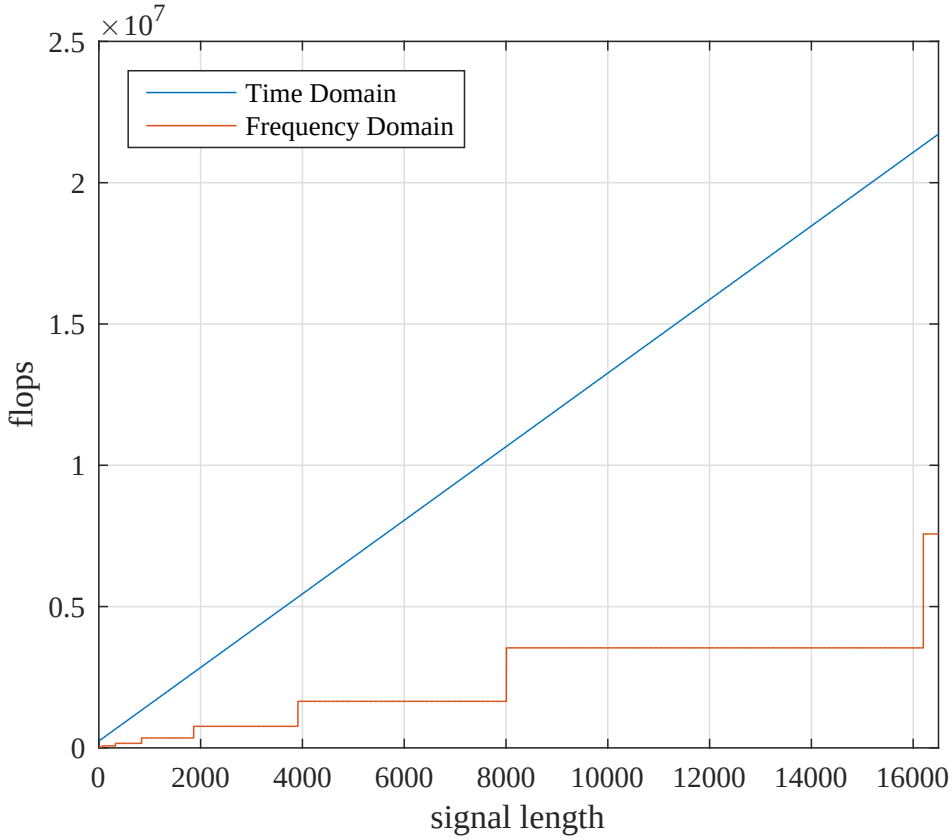



**Figure 6.2:** Comparison of number of floating point operations (flops) required to convolve a varied length complex signal with a 186 tap complex filter.

- frequency domain convolution in a GPU using CUDA libraries

The CPU implements Equation (6.1) in ConvCPU directly on line 209 using a function from lines 11 to 34. The CPU implements Equation (6.2) using the FFTW library on lines 214 to 258.

The GPU implements time domain convolution using global memory in lines 268 to 277. The GPU kernel ConvGPU on lines 36 to 64 is a parallel version of ConvCPU. ConvGPU performs time domain convolution by fetching every element of the signal and filter from global memory.

The GPU implements time domain convolution using shared memory in lines 283 to 292. The GPU kernel ConvGPUshared on lines 67 to 101 is nearly identical to ConvGPU. Threads accessing the same elements of the filter in global memory can be a waste of valuable clock cycles. ConvGPUshared pays and initial price on lines 72 to 76 to move $L_{\text{h}}$ filter coefficients from off chip

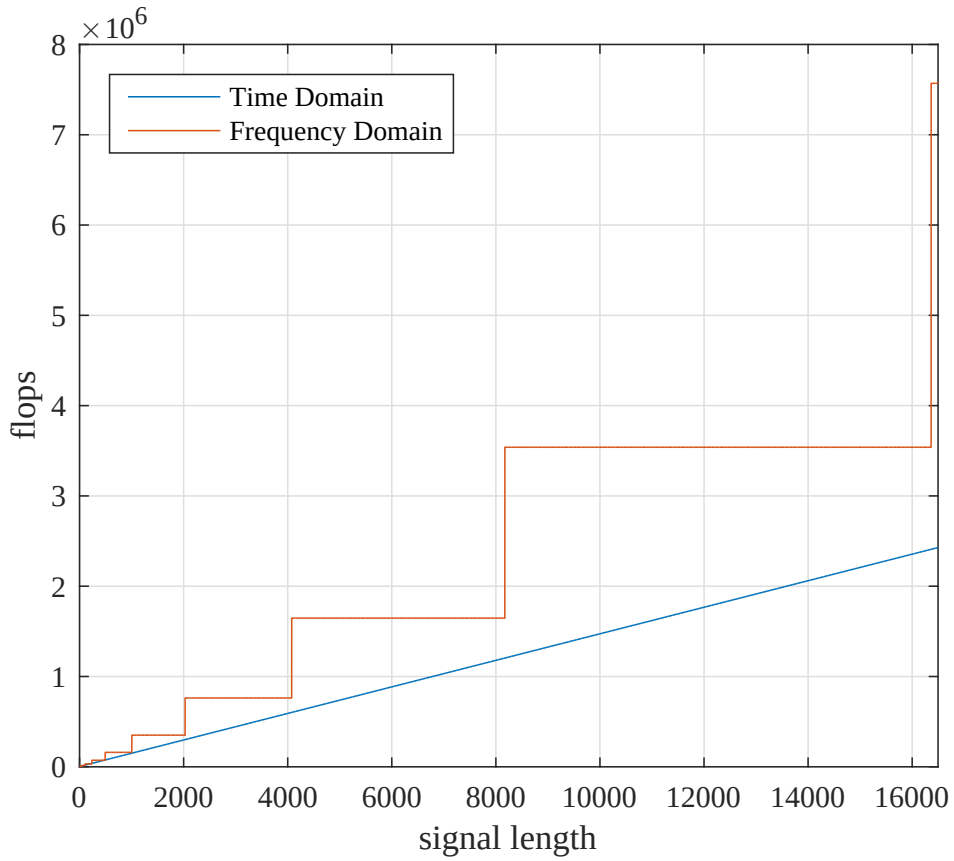



**Figure 6.3:** Comparison of number of floating point operations (flops) required to convolve a varied length complex signal with a 21 tap complex filter.

global memory to the on chip shared memory. Finally, the GPU implements frequency domain convolution using the cuFFT library on lines 298 to 326.

The questions are: Do flops have a direct relationship to execution time on CPUs? Do flops have a direct relationship to execution time on GPUs? When is the initial cost to use shared memory worth it? When should convolution be done in the frequency domain?

The short answer to all of the questions is: GPU execution time depend on the signal length, filter length, CPU, GPU and memory. A good CUDA programmer can make an educated guess on which algorithm may be faster in the GPU, but until all the algorithms have been implemented and timed, there is no definite answer.

To demonstrate that there is no definite answer in GPUs, the execution time of the code in Listing 5.1 was timed. All the memory transfers to and from the host were timed for a fair

**Table 6.1:** Defining start and stop lines for timing comparison in Listing 6.1.

| 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 |

comparison of GPU to CPU. Table 6.1 shows where timing was started and stopped for each convolution implementation.

The execution times shown in Figure 6.4 compares the computation time of a fixed length $186$ tap filter convolved with a varied length signal. The CPU execution time varies enough that the plot is messy. Figure 6.5 shows the lower bounds of execution times by finding the local minimums in $15$ sample windows.

With the plot lower bounded, compare Figure 6.5 to Figure 6.2. Does the CPU and GPU follow the same trend as the number of flops? The CPU has the exact structure that the number of flops predicted. The GPU does have the stair stepping from appending zeros for the frequency domain, but the time domain GPU kernels perform better than the number of flops predicted.

The GPU execution time does not follow the same trend as the number of flops. Why? As mentioned in Section 5.3, GPUs have a ridiculous amount of computational resources and limited memory bandwidth. Over $90\%$ of GPU kernels are memory bandwidth limited. Fast GPU kernels access memory efficiently.

To provide more proof, compare Figures 6.6 and 6.3. Once again, the CPU follows the same trend as the number of flops. The GPU also follows the number of flops trends but to a lesser extent than the CPU. Using shared memory will perform better than using only global memory for "short" filters.

What if the signal length was set and the filter length was varied? Comparing Figure 6.7 to Figure 6.1 shows the CPU follows the trend of flops also. The time domain CPU execution time

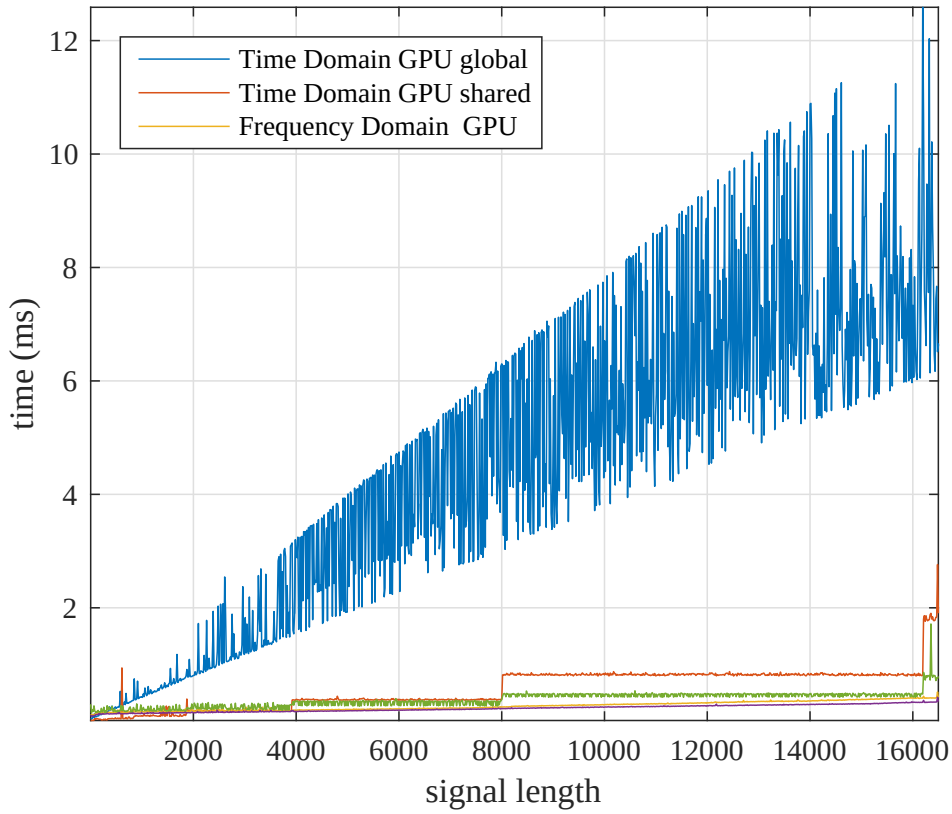



**Figure 6.4:** Comparison of a complex convolution on CPU verse GPU. The signal length is varied and the filter is fixed at 186 taps. The comparison is messy with out lower bounding.

is obviously affected as the number of flops increases. Neither CPU or GPU frequency domain execution time is affected by varying filter length.

The execution time of both time domain GPU convolutions are slightly affected by increasing filter length. The number of memory accesses per output sample increase as the filter length increases. Bottom line, the length of the signal is the largest factor as Equations 6.4 and 6.5 suggest.

For most convolution implementations, the signal and filter lengths are set "magic" numbers. To find the best option, implement convolution every way possible for the given lengths, time each implementation execution time, choose which algorithm is fastest. As Figures 6.4 through 6.7 have shown, unless every implementation is explored, there is no way of saying which implementation will absolutely be fastest.

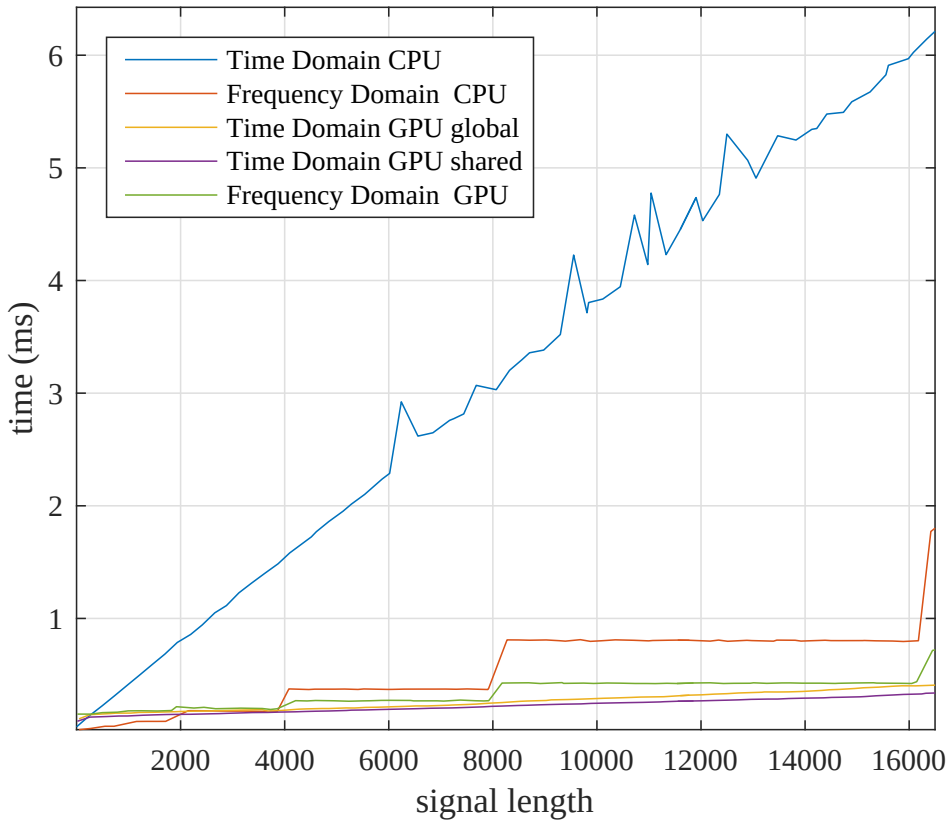



**Figure 6.5:** Comparison of a complex convolution on CPU verse GPU. The signal length is varied and the filter is fixed at 186 taps. A lower bound was applied by searching for a local minimums in 15 sample width windows.

**Table 6.2:** 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 |

Table 6.2 shows the GPU frequency domain algorithm is fastest when convolving a 12672 sample signal with a 186 tap filter. Table 6.3 shows the GPU time domain algorithm using shared memory is fastest when convolving a 12672 sample signal with a 21 tap filter.

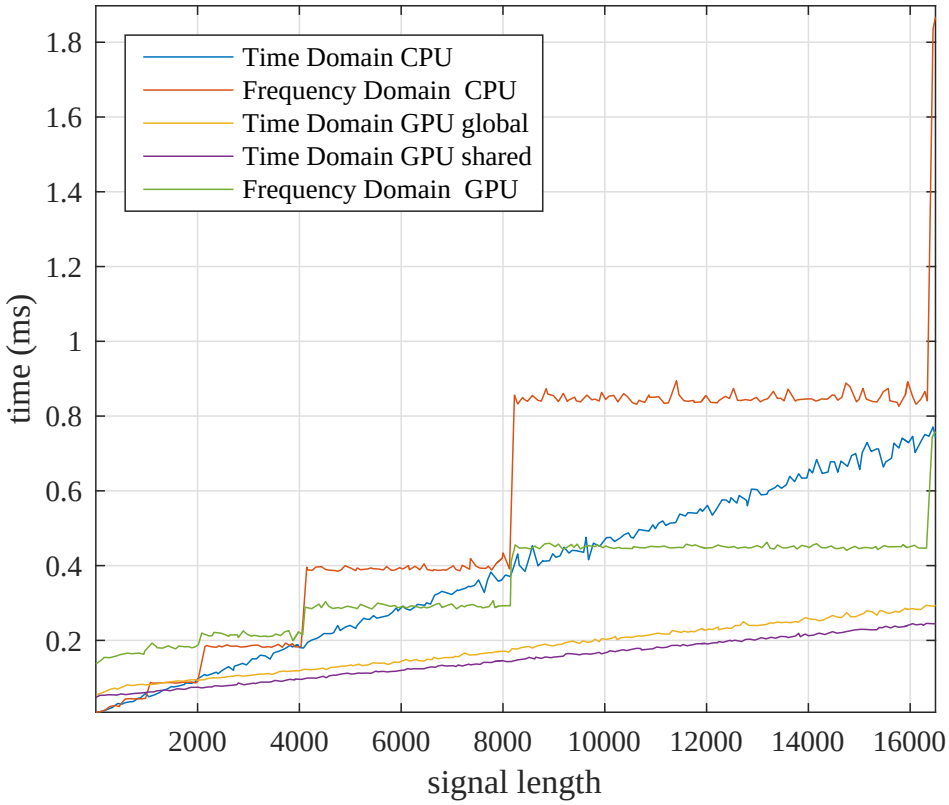



**Figure 6.6:** Comparison of a complex convolution on CPU verse GPU. The signal length is varied and the filter is fixed at 21 taps. A lower bound was applied by searching for a local minimums in 5 sample width windows.

**Table 6.3:** 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 |

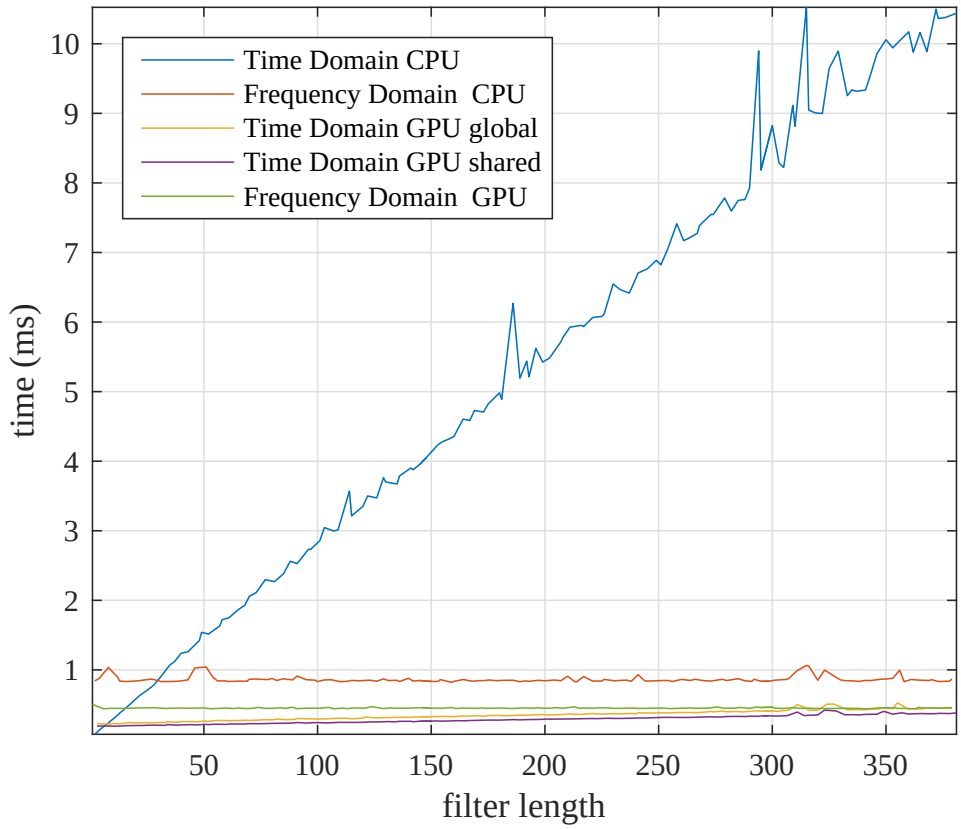



**Figure 6.7:** Comparison of a complex convolution on CPU verse GPU. The filter length is varied and the signal is fixed at 12672 samples. A lower bound was applied by searching for a local minimums in 3 sample width windows.

## 6.2 Batched Convolution

As shown in section 6.1, single convolution doesn't leverage the full power of parallel processing in GPUs. Chapter 3 shows the PAQ received signal has a packetized structure. The received signal has 3104 packets or batches. Batched processing introduces an extra level of parallelism in GPUs because each batch can be processed independently. CUDA has many libraries that are "batched," meaning a GPU kernel is launched for each independent packet or batch of received samples. Some CUDA batched libraries used in PAQ system are cuFFT, cuBLAS and cuSolver.

Batched libraries perform much better than calling a single GPU kernel multiple times with each independent batch of data. Haidar et al. showed batched libaries in GPUs achive more Gflops than calling GPU kernels multiple times [7]. Batched processing performs very well in GPUs

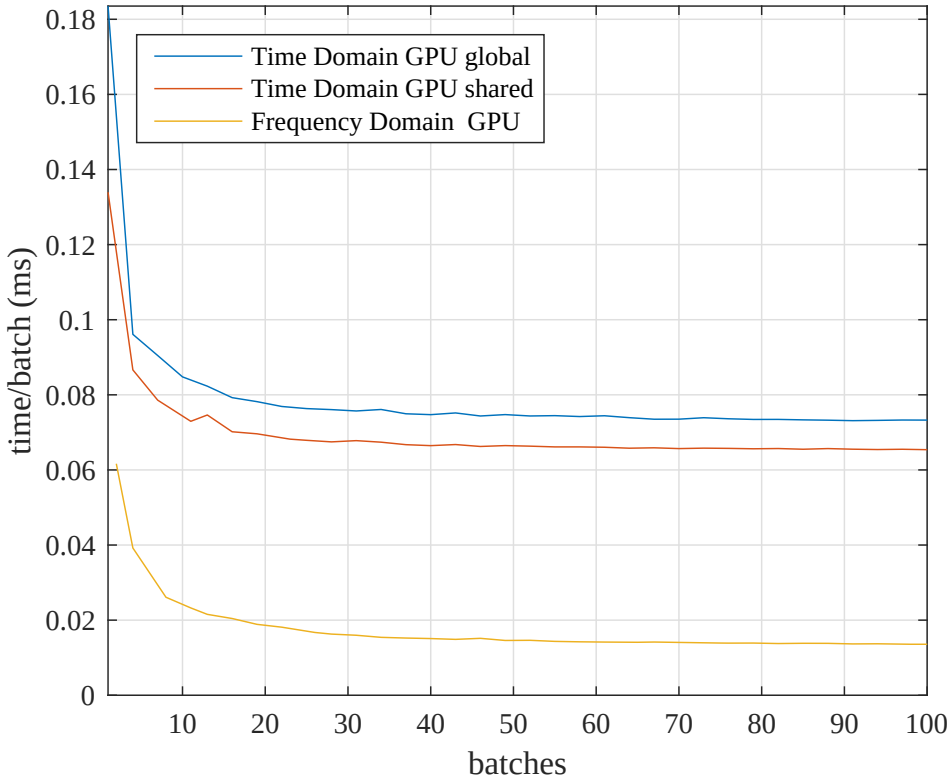



**Figure 6.8:** Comparison on execution time per batch for complex convolution. The number of batches is varied while the signal and filter length is set to 12672 and 186.

because it increases parallelism, reduces overhead on the CPU and provides more opportunities for NVIDIA engineers to optimize.

As the number of batches increases, does CPU and GPU execution time increase linearly? To illistrate how batch processing performs on GPUs, Figure 6.8 shows the execution time per batch for batched convolution in GPUs. The execution time per batch decreases as the number of batches increases but converges after 70 batches.

Figure 6.9 shows how the execution time increases with the number of batches varied. Note that no lower bounding is needed to produce clean batched processing results. This figure shows that frequency domain convolution leverages batch processing better than time domain convolution. No surprise CPU time and frequency domain execution time skyrockets as the number of batches increases.

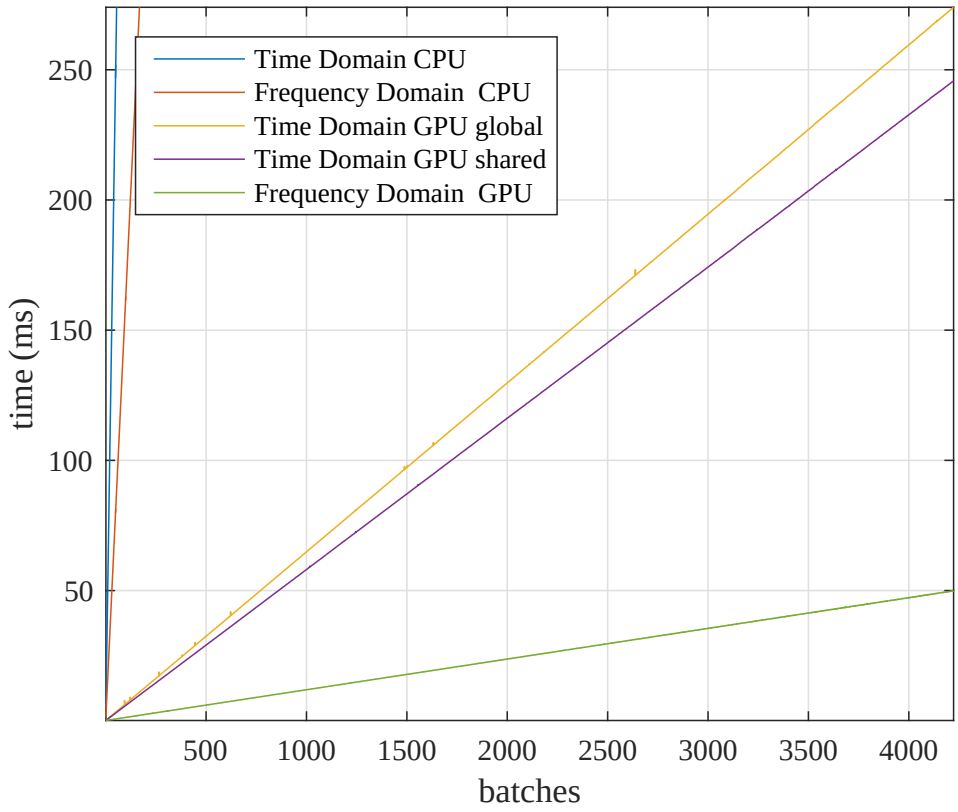



**Figure 6.9:** Comparison of a batched complex convolution on a CPU and GPU. The number of batches is varied while the signal and filter length is set to 12672 and 186.

Judging by Figure 6.9, CPU is not a contender in batched processing compared to the GPU. CPU batched processing will not be explored any further. Listing 6.2 shows three implementations of batched convolution in CUDA

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

Now that the GPU execution time isn't being compared to the CPU, transfers between host and device will not be a factor for algorithm comparison. Table 6.4 shows how Listing 6.2 is timed. Figure 6.10 shows execution time for 3104 batches of 186 tap filters convolved with varying signal lengths. Performing frequency domain convolution is always faster than time domain convolution

**Table 6.4:** Defining start and stop lines for timing comparison in Listing 6.2.

| 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 |

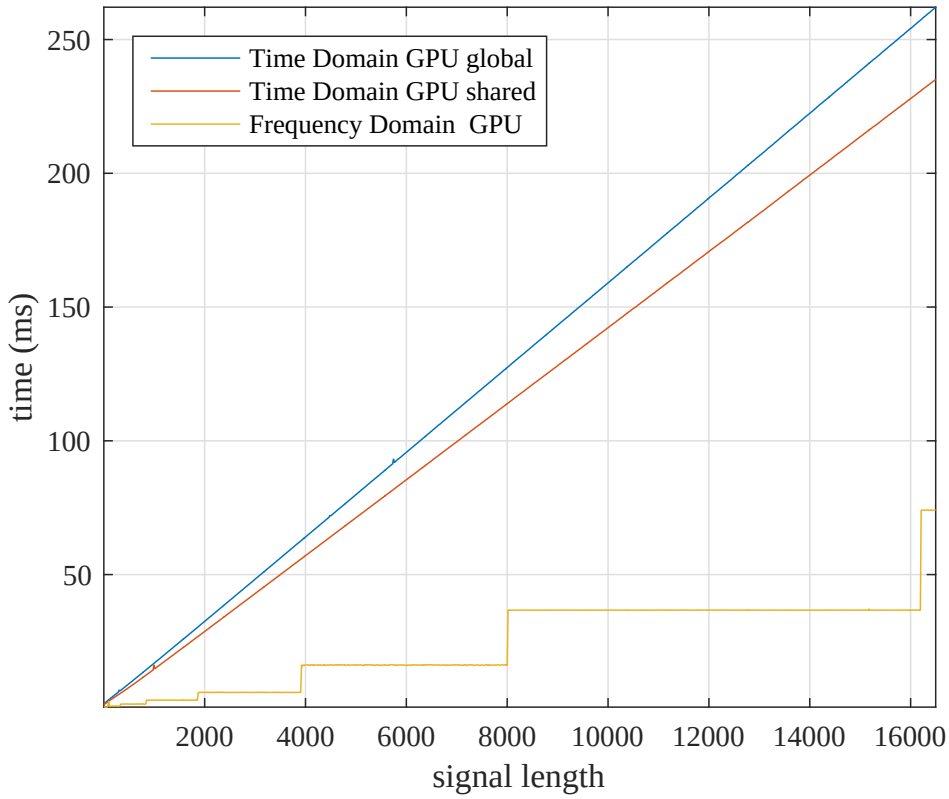



**Figure 6.10:** Comparison of a batched complex convolution on a GPU. The signal length is varied and the filter is fixed at 186 taps.

because the cuFFT library is better optimized for batched processing. The batched GPU implementation of convolution in the Frequency domain convolution takes just 36.8ms for 3104 batches, 12672 sample signals and 186 tap filters. The average exectution time per batch for 3104 batches is 0.0119ms per batch! Compare 0.0119ms per batch to single batched execution time in Table 6.2, one batch took 0.4224. Batched processing introduced a 35$\times$ speed up!

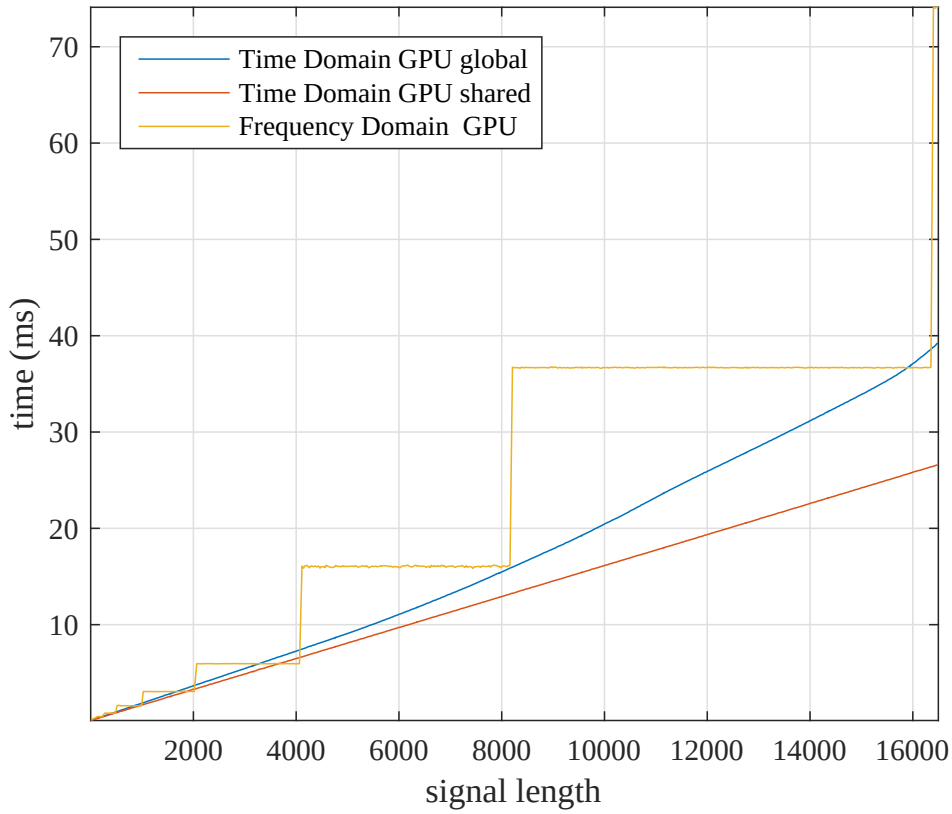



**Figure 6.11:** Comparison of a batched complex convolution on a GPU. The signal length is varied and the filter is fixed at 21 taps.

Figure 6.11 shows execution time for 3104 batches of 21 tap filters convolved with varying signal lengths. This figure exhibits the same characteristics of single batch convolution execution time shown in Figure 6.6. For most signal lengths, performing time domain convolution using shared memory is fastest.

Figure 6.12 shows execution time for 3104 batches of 12672 sample signal convolved with varying filter lengths. This figure exhibits nearly the same characteristics of single batch convolution execution time shown in Figure 6.7 accept the varied filter length has no affect on execution time. For very short filter lengths, time domain convolution using shared memory is fastest. For longer filters , frequency domain convolution is fastest.

Though this section has show the power of batched processing, the algorithm leading to the fastest execution time still depends on signal and filter length, One important concept has been over looked. Figure 4.2 shows there are two filters that need to be applied to the received samples.

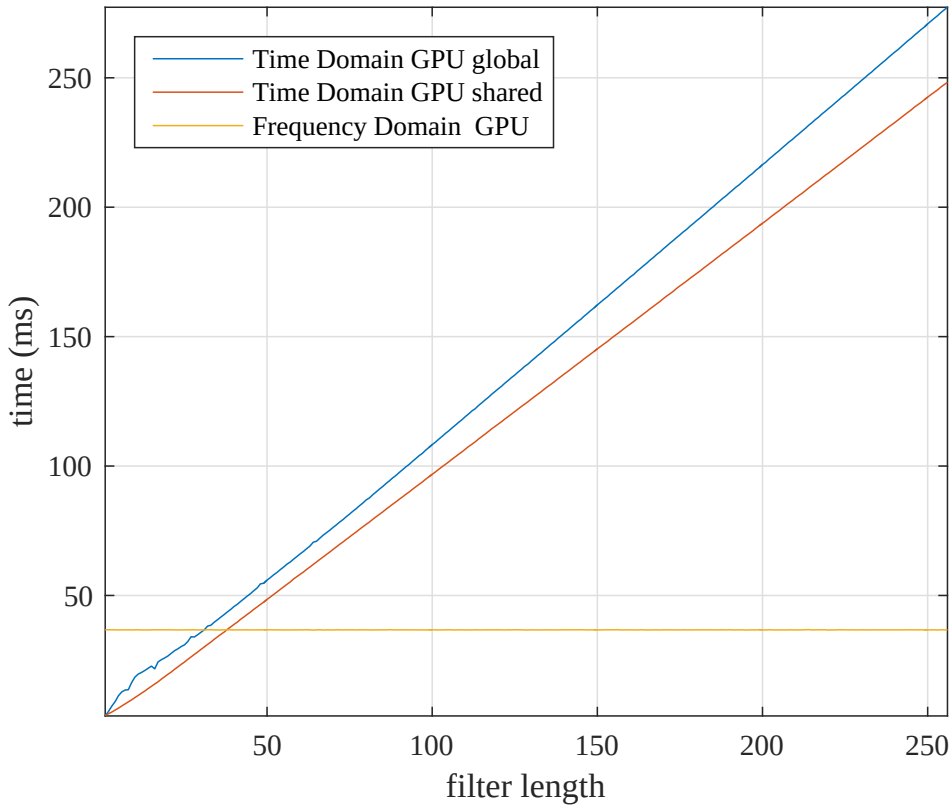



**Figure 6.12:** Comparison of a batched complex convolution on a GPU. The filter length is varied and the signal length is set at 12672 samples.

If convolution is implemented in the time domain, ConvGPU or ConvGPUshared must run twice. The first call of time domain convolution performs an extremely fast "short" convolution of the 186 tap equalizer and 21 detection filter. The second call performs a slower "long" convolution of the 12672 sample signal with the convolved $186 + 21 - 1$ tap filter.

If convolution is implemented in the frequency domain, only the GPU kernel PointToPoint-Multiply has to be updated. PointToPointMultiply is changed from two to three input vectors. For every point the number of memory accesses increases by 1 element and the number of flops doubles from 6 to 12. An extra cuFFT call would be expected accept the detection filter is predefined in Figure 4.2. The FFT of the detection filter is calculated and stored at initialization.

Table 6.5 shows the batched convolution execution time for a 12672 sample signal and 186 tap filter. Table 6.6 shows the batched convolution execution time for a 12672 sample signal and 21

**Table 6.5:** 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 6.6:** 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 |

tap filter. Table 6.7 shows batched cascaded 21 and 186 tap filters convolved with a 12672 sample signal execution time.

Tables 6.5 and 6.6 agree with Figures 6.11 and 6.10. Time domain convolution is faster with a short 21 tap filter but frequency domain convolution is faster with a long 186 tap filter.

Figure 6.13 shows two ways to cascade the signal $\mathbf{r}$ though two filters. The upper blocks apply both filters to the signal taking 180.272ms then 20.4287ms. The lower blocks first convolve the filters to build a $186+21-1$ tap composite filter then apply the 206 tap composite to the signal.

**Table 6.7:** 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 |

**Table 6.8:** Batched convolution execution times with for a 12672 sample signal and 206 tap filter on a Tesla K40c GPU.

| Algorithm | Function or Library | Execution Time (ms) |
|---|---|---|
| GPU time domain global | ConvGPU | 223.064 |
| GPU time domain shared | ConvGPUshared | 199.844 |
| GPU frequency domain | cuFFT | 36.7704 |

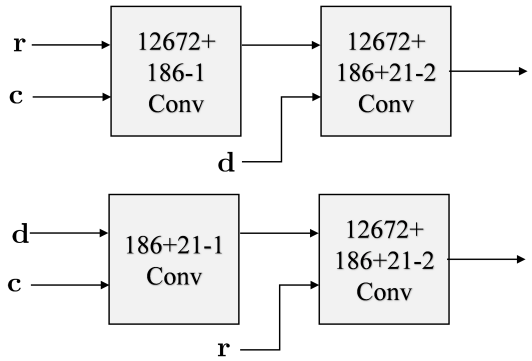



**Figure 6.13:** Two ways to convolve the signal $\mathbf{r}$ with the 186 tap filter $\mathbf{c}$ and 21 tap filter $\mathbf{d}$.

Table 6.7 shows the execution time of implementing cascaded filters, convolving the 21 and 186 tap filters is extremely fast in the GPU. While building the composite filter is extremely fast, time domain convolution suffers a 22.0170ms or 19.7460ms slow down because the composite filter is now 206 taps. Applying an extra filter in the frequency domain only costs 2.3165ms. Table 6.8 confirms it costs an extra 20ms or so to apply a 206 vs 186 tap filter.

**Listing 6.1:** 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 mySignalLength = 1000;
136         int myFilterLength = 186;
137         int myConvLength   = mySignalLength + myFilterLength - 1;
138         int Nfft           = pow(2, ceil(log(myConvLength)/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(mySignalLength*sizeof(cufftComplex));
156         mySignal2               = (cufftComplex*)malloc(Nfft              *sizeof(cufftComplex));
157         mySignal2_fft           = (cufftComplex*)malloc(Nfft          *sizeof(cufftComplex));
158
159         myFilter1               = (cufftComplex*)malloc(myFilterLength*sizeof(cufftComplex));
160         myFilter2               = (cufftComplex*)malloc(Nfft          *sizeof(cufftComplex));
161         myFilter2_fft           = (cufftComplex*)malloc(Nfft              *sizeof(cufftComplex));
162
163         myConv1                 = (cufftComplex*)malloc(myConvLength  *sizeof(cufftComplex));
164         myConv2                 = (cufftComplex*)malloc(Nfft          *sizeof(cufftComplex));
165         myConv2_timeReversed= (cufftComplex*)malloc(Nfft              *sizeof(cufftComplex));
166         myConv3                 = (cufftComplex*)malloc(myConvLength  *sizeof(cufftComplex));
167         myConv4                 = (cufftComplex*)malloc(myConvLength  *sizeof(cufftComplex));
168         myConv5                 = (cufftComplex*)malloc(Nfft          *sizeof(cufftComplex));
169
170         srand(time(0));
171         for(int i = 0; i < mySignalLength; i++){
172                 mySignal1[i].x = rand()%100-50;
173                 mySignal1[i].y = rand()%100-50;
174         }
175
176         for(int i = 0; i < myFilterLength; 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, mySignalLength*sizeof(cufftComplex));
194         cudaMalloc(&dev_mySignal4, mySignalLength*sizeof(cufftComplex));
195         cudaMalloc(&dev_mySignal5, Nfft          *sizeof(cufftComplex));
196
197         cudaMalloc(&dev_myFilter3, myFilterLength*sizeof(cufftComplex));
198         cudaMalloc(&dev_myFilter4, myFilterLength*sizeof(cufftComplex));
199         cudaMalloc(&dev_myFilter5, Nfft          *sizeof(cufftComplex));
200
```

```
201         cudaMalloc(&dev_myConv3,   myConvLength  *sizeof(cufftComplex));
202         cudaMalloc(&dev_myConv4,   myConvLength  *sizeof(cufftComplex));
203         cudaMalloc(&dev_myConv5,   Nfft          *sizeof(cufftComplex));
204
205
206         /**
207          * Time Domain Convolution CPU
208          */
209         ConvCPU(myConv1,mySignal1,myFilter1,mySignalLength,myFilterLength);
210
211         /**
212          * Frequency Domain Convolution CPU
213          */
214         fftwf_plan forwardPlanSignal = fftwf_plan_dft_1d(Nfft, (fftwf_complex*)mySignal2,    (
                fftwf_complex*)mySignal2_fft,        FFTW_FORWARD, FFTW_MEASURE);
215         fftwf_plan forwardPlanFilter = fftwf_plan_dft_1d(Nfft, (fftwf_complex*)myFilter2,
                (fftwf_complex*)myFilter2_fft,           FFTW_FORWARD, FFTW_MEASURE);
216         fftwf_plan backwardPlanConv  = fftwf_plan_dft_1d(Nfft, (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 < Nfft; i++){
220                 if(i<mySignalLength)
221                         mySignal2[i] = mySignal1[i];
222                 else
223                         mySignal2[i] = zero;
224
225                 if(i<myFilterLength)
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 < Nfft; 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 < Nfft; i++){
252                 result.x = myConv2_timeReversed[Nfft-i].x/Nfft;
253                 result.y = myConv2_timeReversed[Nfft-i].y/Nfft;
254                 myConv2[i] = result;
255         }
256         result.x = myConv2_timeReversed[0].x/Nfft;
257         result.y = myConv2_timeReversed[0].y/Nfft;
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)*mySignalLength,
               cudaMemcpyHostToDevice);
269        cudaMemcpy(dev_myFilter3, myFilter1, sizeof(cufftComplex)*myFilterLength,
               cudaMemcpyHostToDevice);
270
271        int numTreadsPerBlock = 512;
272        int numBlocks = myConvLength/numTreadsPerBlock;
273        if(myConvLength % numTreadsPerBlock > 0)
274                numBlocks++;
275        ConvGPU<<<numBlocks, numTreadsPerBlock>>>(dev_myConv3, dev_mySignal3, dev_myFilter3,
               mySignalLength, myFilterLength);
276
277        cudaMemcpy(myConv3, dev_myConv3, myConvLength*sizeof(cufftComplex),
               cudaMemcpyDeviceToHost);
278
279
280        /**
281         * Time Domain Convolution GPU Using Shared Memory
282         */
283        cudaMemcpy(dev_mySignal4, mySignal1, sizeof(cufftComplex)*mySignalLength,
               cudaMemcpyHostToDevice);
284        cudaMemcpy(dev_myFilter4, myFilter1, sizeof(cufftComplex)*myFilterLength,
               cudaMemcpyHostToDevice);
285
286        numTreadsPerBlock = 512;
287        numBlocks = myConvLength/numTreadsPerBlock;
288        if(myConvLength % numTreadsPerBlock > 0)
289                numBlocks++;
290        ConvGPUshared<<<numBlocks, numTreadsPerBlock,myFilterLength*sizeof(cufftComplex)>>>(
               dev_myConv4, dev_mySignal4, dev_myFilter4, mySignalLength, myFilterLength);
291
292        cudaMemcpy(myConv4, dev_myConv4, myConvLength*sizeof(cufftComplex),
               cudaMemcpyDeviceToHost);
293
294
295        /**
296         * Frequency Domain Convolution GPU
297         */
298        cufftHandle plan;
299        int n[1] = {Nfft};
300        cufftPlanMany(&plan,1,n,NULL,1,1,NULL,1,1,CUFFT_C2C,1);
301
302        cudaMemset(dev_mySignal5, 0,         Nfft*sizeof(cufftComplex));
303        cudaMemset(dev_myFilter5, 0,         Nfft*sizeof(cufftComplex));
304
305        cudaMemcpy(dev_mySignal5, mySignal2, Nfft*sizeof(cufftComplex), cudaMemcpyHostToDevice);
306        cudaMemcpy(dev_myFilter5, myFilter2, Nfft*sizeof(cufftComplex), cudaMemcpyHostToDevice);
307
308        cufftExecC2C(plan, dev_mySignal5, dev_mySignal5, CUFFT_FORWARD);
309        cufftExecC2C(plan, dev_myFilter5, dev_myFilter5, CUFFT_FORWARD);
310
311        numTreadsPerBlock = 512;
312        numBlocks = Nfft/numTreadsPerBlock;
313        if(Nfft % numTreadsPerBlock > 0)
314                numBlocks++;
315        PointToPointMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal5, dev_myFilter5, Nfft
               );
316
317        cufftExecC2C(plan, dev_mySignal5, dev_mySignal5, CUFFT_INVERSE);
318
319        numTreadsPerBlock = 128;
320        numBlocks = Nfft/numTreadsPerBlock;
321        if(Nfft % numTreadsPerBlock > 0)
322                numBlocks++;
323        float scalar = 1.0/((float)Nfft);
324        ScalarMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal5, scalar, Nfft);
```

```
325
326         cudaMemcpy(myConv5, dev_mySignal5, Nfft*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 6.2:** 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 mySignalLength = 12672;
116         int myFilterLength = 186;
117         int myConvLength   = mySignalLength + myFilterLength - 1;
118         int Nfft           = pow(2, ceil(log(myConvLength)/log(2)));
119         int maxThreads;
120         int numTreadsPerBlock;
121         int numBlocks;
122
123         cufftHandle plan;
124         int n[1] = {Nfft};
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(mySignalLength*numBatches*sizeof(cufftComplex));
136         mySignal1_pad  = (cufftComplex*) malloc(Nfft             *numBatches*sizeof(cufftComplex
                ));
137         myFilter1      = (cufftComplex*) malloc(myFilterLength*numBatches*sizeof(cufftComplex));
138         myFilter1_pad  = (cufftComplex*) malloc(Nfft          *numBatches*sizeof(cufftComplex));
139         myConv1        = (cufftComplex*) malloc(myConvLength  *numBatches*sizeof(cufftComplex));
140         myConv2        = (cufftComplex*) malloc(myConvLength  *numBatches*sizeof(cufftComplex));
141         myConv3        = (cufftComplex*) malloc(Nfft             *numBatches*sizeof(cufftComplex
                ));
142
143         srand(time(0));
144         for(int i = 0; i < mySignalLength; i++){
145                 mySignal1[i].x = rand()%100-50;
146                 mySignal1[i].y = rand()%100-50;
147         }
148
149         for(int i = 0; i < myFilterLength; 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<Nfft*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 < mySignalLength; i++){
163                         mySignal1[batch*mySignalLength+i] = mySignal1[i];
164                         mySignal1_pad[batch*Nfft+i] = mySignal1[i];
165                 }
166                 for(int i = 0; i < myFilterLength; i++){
167                         myFilter1[batch*myFilterLength+i] = myFilter1[i];
168                         myFilter1_pad[batch*Nfft+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, mySignalLength*numBatches*sizeof(cufftComplex));
183         cudaMalloc(&dev_mySignal2, mySignalLength*numBatches*sizeof(cufftComplex));
184         cudaMalloc(&dev_mySignal3, Nfft             *numBatches*sizeof(cufftComplex));
185         cudaMalloc(&dev_myFilter1, myFilterLength*numBatches*sizeof(cufftComplex));
186         cudaMalloc(&dev_myFilter2, myFilterLength*numBatches*sizeof(cufftComplex));
187         cudaMalloc(&dev_myFilter3, Nfft                 *numBatches*sizeof(cufftComplex));
188         cudaMalloc(&dev_myConv1,   myConvLength  *numBatches*sizeof(cufftComplex));
189         cudaMalloc(&dev_myConv2,   myConvLength  *numBatches*sizeof(cufftComplex));
190         cudaMalloc(&dev_myConv3,   Nfft                 *numBatches*sizeof(cufftComplex));
191
192         /**
193          * Time Domain Convolution GPU Using Global Memory
194          */
195         cudaMemcpy(dev_mySignal1, mySignal1, numBatches*sizeof(cufftComplex)*mySignalLength,
                cudaMemcpyHostToDevice);
```

```
196         cudaMemcpy(dev_myFilter1, myFilter1, numBatches*sizeof(cufftComplex)*myFilterLength,
                cudaMemcpyHostToDevice);
197
198         maxThreads = myConvLength*numBatches;
199         numTreadsPerBlock = 128;
200         numBlocks = maxThreads/numTreadsPerBlock;
201         if(maxThreads % numTreadsPerBlock > 0)
202                 numBlocks++;
203         ConvGPU<<<numBlocks, numTreadsPerBlock>>>(dev_myConv1, dev_mySignal1, dev_myFilter1,
                mySignalLength, myFilterLength, maxThreads);
204
205         cudaMemcpy(myConv1, dev_myConv1, myConvLength*numBatches*sizeof(cufftComplex),
                cudaMemcpyDeviceToHost);
206
207         /**
208          * Time Domain Convolution GPU Using Shared Memory
209          */
210         cudaMemcpy(dev_mySignal2, mySignal1, numBatches*sizeof(cufftComplex)*mySignalLength,
                cudaMemcpyHostToDevice);
211         cudaMemcpy(dev_myFilter2, myFilter1, numBatches*sizeof(cufftComplex)*myFilterLength,
                cudaMemcpyHostToDevice);
212
213         maxThreads = myConvLength*numBatches;
214         numTreadsPerBlock = 256;
215         numBlocks = maxThreads/numTreadsPerBlock;
216         if(maxThreads % numTreadsPerBlock > 0)
217                 numBlocks++;
218         ConvGPUshared<<<numBlocks, numTreadsPerBlock, myFilterLength*sizeof(cufftComplex)>>>(
                dev_myConv2, dev_mySignal2, dev_myFilter2, mySignalLength, myFilterLength,maxThreads)
                ;
219
220         cudaMemcpy(myConv2, dev_myConv2, myConvLength*numBatches*sizeof(cufftComplex),
                cudaMemcpyDeviceToHost);
221
222         /**
223          * Frequency Domain Convolution GPU
224          */
225         cudaMemcpy(dev_mySignal3, mySignal1_pad, Nfft*numBatches*sizeof(cufftComplex),
                cudaMemcpyHostToDevice);
226         cudaMemcpy(dev_myFilter3, myFilter1_pad, Nfft*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 = Nfft*numBatches;
232         numTreadsPerBlock = 96;
233         numBlocks = maxThreads/numTreadsPerBlock;
234         if(maxThreads % numTreadsPerBlock > 0)
235                 numBlocks++;
236         PointToPointMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal3, dev_myFilter3,
                maxThreads);
237         cufftExecC2C(plan, dev_mySignal3, dev_mySignal3, CUFFT_INVERSE);
238
239         numTreadsPerBlock = 640;
240         numBlocks = maxThreads/numTreadsPerBlock;
241         if(maxThreads % numTreadsPerBlock > 0)
242                 numBlocks++;
243         float scalar = 1.0/((float)Nfft);
244         ScalarMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal3, scalar, maxThreads);
245
246         cudaMemcpy(myConv3, dev_mySignal3, Nfft*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 7

# Equalizer Equations

This thesis examines that performance and GPU implementation of 5 equalizers. 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)
- Minimum Mean Square Error (MMSE)
- Constant Modulus Algorithm (CMA)
- Frequency Domain Equalizer 1 (FDE1)
- Frequency Domain Equalizer 2 (FDE2)

The ZF and MMSE equalizers are very similar in formulation though from different sources. As you might tell from the names, FDE1 and FDE2 are also very similar with one subtle difference. CMA isn't related to any other algorithm aside from being initialized by MMSE.

## 7.1 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

$$\mathbf{R}\mathbf{c} = \hat{\mathbf{h}} \tag{7.1}$$

where $\mathbf{c}$ is a vector of desired equalizer coefficients and $\mathbf{R}$ is an auto-correlation matrix of the channel estimate $\hat{\mathbf{h}}$. It will be shown that the only difference between ZF and MMSE lies in the the auto-correlation matrix $\mathbf{R}$.

### 7.1.1 Zero-Forcing

The ZF equalizer is an FIR filter defined by the coefficients

$$c_{\text{ZF}}(-L_1) \quad \cdots \quad c_{\text{ZF}}(0) \quad \cdots \quad c_{\text{ZF}}(L_2). \tag{7.2}$$

The filter coefficients are the solution to the matrix vector equation [10, eq. (311)]

$$\mathbf{c}_{\text{ZF}} = \left(\mathbf{H}^{\dagger}\mathbf{H}\right)^{-1}\mathbf{H}^{\dagger}\mathbf{u}_{n_0} \tag{7.3}$$

where

$$\mathbf{c}_{\text{ZF}} = \begin{bmatrix} c_{\text{ZF}}(-L_1) \\ \vdots \\ c_{\text{ZF}}(0) \\ \vdots \\ c_{\text{ZF}}(L_2) \end{bmatrix}, \tag{7.4}$$

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \Big\} n_0 - 1 \text{ zeros} \\ \begin{bmatrix} 1 \end{bmatrix} \\ \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} \Big\} N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \tag{7.5}$$

where $n_0 = N_1 + L_1 + 1$ and

$$\mathbf{H} = \begin{bmatrix} \hat{h}(-N_1) & & & \\ \hat{h}(-N_1+1) & \hat{h}(-N_1) & & \\ \vdots & \vdots & \ddots & \\ \hat{h}(N_2) & \hat{h}(N_2-1) & & \hat{h}(-N_1) \\ & \hat{h}(N_2) & & \hat{h}(-N_1+1) \\ & & & \vdots \\ & & & \hat{h}(N_2) \end{bmatrix}. \tag{7.6}$$

Equation (7.3) 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 (7.3) results in

$$\begin{aligned} \mathbf{H}^\dagger\mathbf{H}\mathbf{c}_{\text{ZF}} &= \mathbf{H}^\dagger\mathbf{u}_{n_0} \\ \mathbf{R}_{\hat{h}}\mathbf{c}_{\text{ZF}} &= \hat{\mathbf{h}}_{n_0} \end{aligned} \tag{7.7}$$

where

$$\mathbf{R}_{\hat{h}} = \mathbf{H}^\dagger\mathbf{H} = \begin{bmatrix} r_{\hat{h}}(0) & r^*_{\hat{h}}(1) & \cdots & r^*_{\hat{h}}(L_{eq}-1) \\ r_{\hat{h}}(1) & r_{\hat{h}}(0) & \cdots & r^*_{\hat{h}}(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_{\hat{h}}(L_{eq}-1) & r_{\hat{h}}(L_{eq}-2) & \cdots & r_{\hat{h}}(0) \end{bmatrix} \tag{7.8}$$

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

$$\hat{\mathbf{h}}_{n_0} = \mathbf{H}^{\dagger}\mathbf{u}_{n_0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix} \tag{7.9}$$

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

$$r_{\hat{h}}(k) = \sum_{n=-N_1}^{N_2} \hat{h}(n)\hat{h}^*(n-k). \tag{7.10}$$

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

$$\mathbf{r}_{\hat{h}} = \begin{bmatrix} r_{\hat{h}}(0) \\ \vdots \\ r_{\hat{h}}(L_{ch}) \\ r_{\hat{h}}(L_{ch}+1) \\ \vdots \\ r_{\hat{h}}(L_{eq}-1) \end{bmatrix} = \begin{bmatrix} r_{\hat{h}}(0) \\ \vdots \\ r_{\hat{h}}(L_{ch}) \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{7.11}$$

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.

### 7.1.2 MMSE Equalizer

The MMSE equalizer is an FIR filter defined by the coefficients

$$c_{\text{MMSE}}(-L_1) \quad \cdots \quad c_{\text{MMSE}}(0) \quad \cdots \quad c_{\text{MMSE}}(L_2). \tag{7.12}$$

The filter coefficients are the solution to the matrix vector equation [10, eq. (330) and (333)]

$$\mathbf{c}_{MMSE} = \left[\mathbf{G}\mathbf{G}^{\dagger} + 2\hat{\sigma}_w^2 \mathbf{I}_{L_1+L_2+1}\right]^{-1} \mathbf{g}^{\dagger} \tag{7.13}$$

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

$$\mathbf{G} = \begin{bmatrix} \hat{h}(N_2) & \cdots & \hat{h}(-N_1) & & & \\ & \hat{h}(N_2) & \cdots & \hat{h}(-N_1) & & \\ & & \ddots & & \ddots & \\ & & & \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \end{bmatrix} \tag{7.14}$$

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

$$\mathbf{g}^{\dagger} = \hat{\mathbf{h}}_{n0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix}. \tag{7.15}$$

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 (7.8). To further simplify MMSE, twice the estimated noise variance is added down the diagonal

of the channel estimate auto-correlation matrix

$$\mathbf{R}_{\hat{h}w} = \mathbf{R}_{\hat{h}} + 2\hat{\sigma_w^2}\mathbf{I}_{L_1+L_2+1} = \begin{bmatrix} r_h(0) + 2\hat{\sigma_w^2} & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) + 2\hat{\sigma_w^2} & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) + 2\hat{\sigma_w^2} \end{bmatrix}. \tag{7.16}$$

By placing Equation (7.16) and (7.15) into (7.13) results in

$$\mathbf{c}_{\text{MMSE}} = \mathbf{R}_{\hat{h}w}^{-1}\hat{\mathbf{h}}_{n_0}. \tag{7.17}$$

Solving for the MMSE equalizer coefficients $\mathbf{c}_{\text{MMSE}}$ takes the form like the ZF equalizer coeffiencts in (7.7)

$$\mathbf{R}_{\hat{h}w}\mathbf{c}_{\text{MMSE}} = \hat{\mathbf{h}}_{n_0}. \tag{7.18}$$

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

## 7.2 The Constant Modulus Algorithm

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

$$c_{\text{CMA}(b)}(-L_1) \quad \cdots \quad c_{\text{CMA}(b)}(0) \quad \cdots \quad c_{\text{CMA}(b)}(L_2). \tag{7.19}$$

The filter coefficients are calculated by a steepest decent algorithm

$$\mathbf{c}_{\text{CMA}(b+1)} = \mathbf{c}_{\text{CMA}(b)} - \mu\nabla J \tag{7.20}$$

initialized by the MMSE equalizer coefficients

$$\mathbf{c}_{\text{CMA(0)}} = \mathbf{c}_{\text{MMSE}}. \tag{7.21}$$

The vector $\mathbf{J}$ is the cost function and $\nabla J$ is the cost function gradient [10, eq. (352)]

$$\nabla J = \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \Big[ y(n)y^*(n) - 1 \Big] \, y(n)\mathbf{r}^*(n). \tag{7.22}$$

where

$$\mathbf{r}(n) = \begin{bmatrix} r(n+L_1) \\ \vdots \\ r(n) \\ \vdots \\ r(n-L_2) \end{bmatrix}. \tag{7.23}$$

This means $\nabla J$ is defined by

$$\nabla J = \begin{bmatrix} \nabla J(-L_1) \\ \vdots \\ \nabla J(0) \\ \vdots \\ \nabla J(L_2) \end{bmatrix}. \tag{7.24}$$

A DSP engineer could implement the steepest decent algorithm by computing the cost function gradient directly. The long summation for $\nabla J$ in (7.22) does not map well to GPUs. Later chapters will show how well convolution performs on GPUs. The computation for $\nabla J$ can be massaged and re-expressed as convolution.

To begin messaging $\nabla J$, the term

$$z(n) = 2 \Big[ y(n)y^*(n) - 1 \Big] \, y(n) \tag{7.25}$$

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

$$\nabla J = \frac{1}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} z(n)\mathbf{r}^*(n). \tag{7.26}$$

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

$$\nabla J \;=\; \frac{z(0)}{L_{pkt}} \begin{bmatrix} r^*(L_1) \\ \vdots \\ r^*(0) \\ \vdots \\ r^*(L_2) \end{bmatrix} + \frac{z(1)}{L_{pkt}} \begin{bmatrix} r^*(1+L_1) \\ \vdots \\ r^*(1) \\ \vdots \\ r^*(1-L_2) \end{bmatrix} + \cdots \frac{z(L_{pkt}-1)}{L_{pkt}} \begin{bmatrix} r^*(L_{pkt}-1+L_1) \\ \vdots \\ r^*(L_{pkt}-1) \\ \vdots \\ r^*(L_{pkt}-1-L_2) \end{bmatrix} \tag{7.27}$$

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

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-k), \quad -L_1 \le k \le L_2. \tag{7.28}$$

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

$$\rho(n) = r^*(n). \tag{7.29}$$

Now

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)\rho(k-m). \tag{7.30}$$

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

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

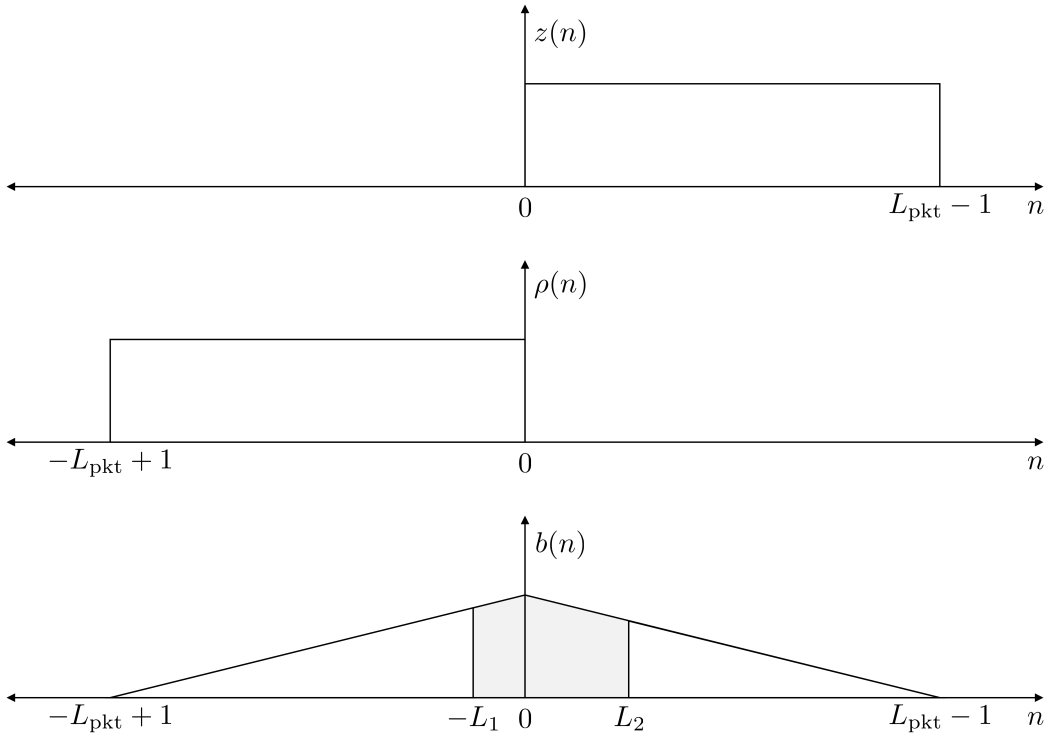



**Figure 7.1:** Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.

$$= \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-n) \tag{7.31}$$

Comparing Equation (7.30) and (7.31) shows that

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2. \tag{7.32}$$

The values of interest are shown in Figure 7.1.

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

**Table 7.1:** 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
```

## 7.3 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's Frequency Domain Equalizers [11, eq. (11) and (12)].

### 7.3.1 Frequency Domain Equalizer One

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

$$C_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \cdots, L-1. \tag{7.33}$$

The term $C_{\text{FDE1}}(e^{j\omega_k})$ is FDE1s frequency response at $\omega_k$. The term $\hat{H}(e^{j\omega_k})$ is the channel estimate frequency response at $\omega_k$. The term $\hat{\sigma}^2$ is the noise variance estimate, this term is completely independent of frequency because the noise is assumed to be spectrally flat or white.

Equation (7.33) is straight forward to implement in GPUs. FDE1 is extremely fast and computationally efficient.

### 7.3.2 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 [11, eq. (12)]. The frequency respoonse of FDE2 is

$$C_{\text{FDE2}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \cdots, L-1 \tag{7.34}$$

where $\Psi(e^{j\omega_k})$ is show in Figure 7.2. The term $\Psi(e^{j\omega_k})$ eliminates out of band multipath that may be challenging to estimate and over come.

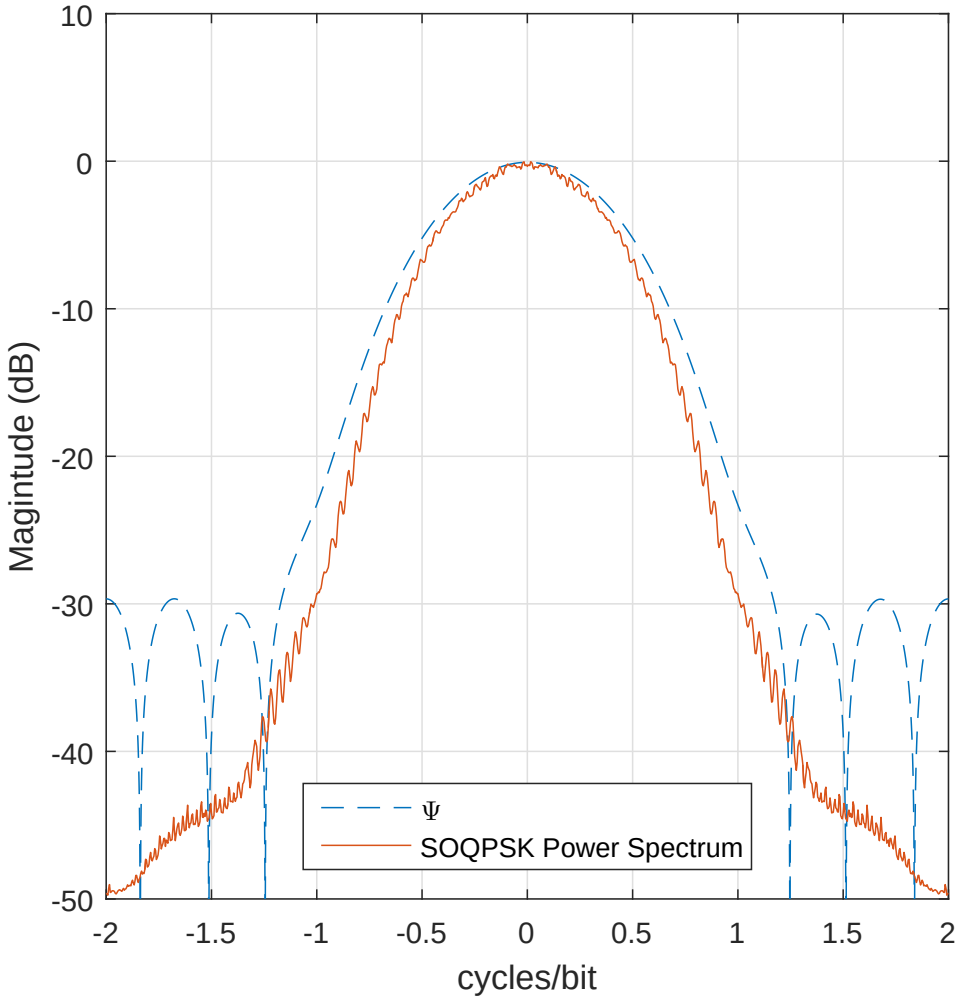



**Figure 7.2:** I need help on this one!!!!

# Chapter 8

# Equalizer GPU Implementation

Each equalizer presents an interesting challenge from a GPU implementation perspective. This is why in Chapter 7 the equalizer equations where massaged and conditioned. This chapter will explain how the equalizers were computed and applied in batch processing.

Chapter 6 showed the true power of batched processing in GPUs. Each batch of data is totally independent of other batches. For this reason and to simplify figures, assume every block diagram in this chapter is duplicated $3104$ times on the GPU. If a block diagram shows how to compute one equalizer coefficients vector, assume that the block diagram is repeated $3104$ times to compute $3104$ equalizer coefficent vectors.

Convolution is used many times in this chapter. Frequency domain convolution has many little steps that make a block diagram look busy. Figures 8.1 and 8.2 show how frequency domain convolution will be represented as one block in this chapter.

## 8.1 Zero-Forcing and MMSE GPU Implementation

The ZF and MMSE equalizer coefficient computations have exactly the same form as shown in Equations 7.7 and 7.18

$$\mathbf{R}_{\hat{h}}\mathbf{c}_{\text{ZF}} = \hat{\mathbf{h}}_{n_0} \tag{8.1}$$

$$\mathbf{R}_{\hat{h}w}\mathbf{c}_{\text{MMSE}} = \hat{\mathbf{h}}_{n_0}. \tag{8.2}$$

The only difference is $\mathbf{R}_{\hat{h}}$ in ZF and $\mathbf{R}_{\hat{h}w}$ in MMSE. Computing the ZF and MMSE equalizer coefficients is extremely computationally heavy.

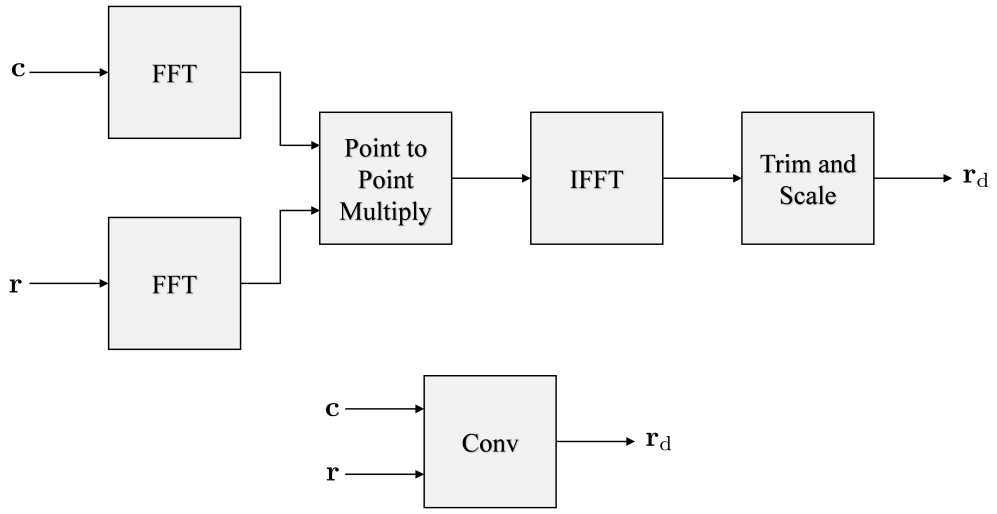



**Figure 8.1:** Convolution of vectors $\mathbf{c}$ and $\mathbf{r}$ block diagram simplified to one block marked Conv.

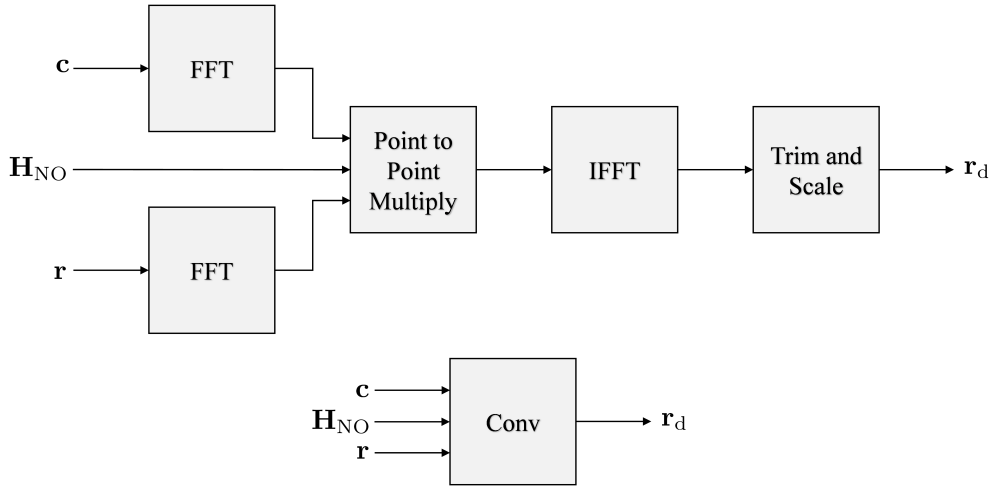



**Figure 8.2:** Convolution of vectors $\mathbf{c}$, $\mathbf{r}$ and $\mathbf{H}_\mathrm{NO}$ block diagram simplified to one block marked Conv.

It is possible to obtain the ZF and MMSE equalizer coefficients by computing the inverse of

$$\mathbf{c}_\mathrm{ZF} = \mathbf{R}_{\hat{h}}^{-} 1 \hat{\mathbf{h}}_{n_0} \tag{8.3}$$

$$\mathbf{c}_\mathrm{MMSE} = \mathbf{R}_{\hat{h}w}^{-} 1 \hat{\mathbf{h}}_{n_0}. \tag{8.4}$$

Before solving Equation 8.3, $\mathbf{R}_{\hat{h}}$ and $\hat{\mathbf{h}}_{n_0}$ need to be built and calculated given $\hat{\mathbf{h}}$. The matrix $\mathbf{R}_{\hat{h}}$ requires the sample auto-correlation of the estimated channel $\mathbf{r}_{\hat{h}}$ and the time reversed channel and shifted channel $\hat{\mathbf{h}}_{n_0}$.

The vector $\hat{\mathbf{h}}_{n_0}$ is just the time reversed and conjugated estimated channel $\hat{\mathbf{h}}$. Building $\hat{\mathbf{h}}_{n_0}$ is trivial in the GPU and very little optimizing needs to be performed. Computing the sample auto-correlation $\mathbf{r}_{\hat{h}}$ is done by implementing Equation 7.10 directly in one GPU kernel. The computation of $\mathbf{r}_{\hat{h}}$ is very fast because the length of the channel estimate $L_{\text{ch}}$ is very short.

Section 7.1.1 showed that $\mathbf{R}_{\hat{h}}$ is sparse because $r_{\hat{h}}(k)$ only has support on $-L_{\text{ch}} \leq k \leq L_{\text{ch}}$. To reduce computation time, the sparseness of $\mathbf{R}_{\hat{h}}$ will be leveraged. With out leveraging the sparse properties of $\mathbf{R}_{\hat{h}}$, even the mighty Tesla K40c cannot produce $\mathbf{c}_{\text{ZF}}$ in less than $500$ms.

The sparseness of $\mathbf{R}_{\hat{h}}$ is leveraged using a sparse solver function called "cusolverSpCcsrqrsvBatched". cusolverSpCcsrqrsvBatched is a batched complex solver that leverages the sparse properties of $\mathbf{R}_{\hat{h}}$ by utilizing Compressed Row Storage (CRS) [12]. The Compressed Row Storage reduces a large $186 \times 186$ matrix $\mathbf{R}_{\hat{h}}$ to a $12544$ element CSR matrix $\mathbf{R}_{\hat{h}\text{CRS}}$. 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 [6].

Figure 8.3 shows how the Zero-Forcing equalizer coefficients are implemented in the GPU. The MMSE equalizer coefficients are computed nearly identically to ZF accept when calculating $\mathbf{R}_{\hat{h}w\text{CRS}}$, $\hat{\sigma}_w^2$ is added to the main diagonal elements.

Using cusolverSpCcsrqrsvBatched wasn't the only implementation researched. Table 8.1 lists the algorithms researched and their respective execution times. A custom GPU Levinson Recursion algorithm was built to leverage the Toeplitz structure of $\mathbf{R}_{\hat{h}}$ and $\mathbf{R}_{\hat{h}w}$ [13, Chap. 5]. The Levinson Recursion algorithm initially showed promise when operating on real floats but when converted to complex data Levinson wasn't feasable.

Rather than solving for $\mathbf{c}_{\text{ZF}}$ or $\mathbf{c}_{\text{MMSE}}$, computing the full inverse of $\mathbf{R}_{\hat{h}}$ was researched using the cuBLAS LU Decomposition. While using staying real time using cuBLAS was feasable, cusolverSp out performed cuBLAS by almost $2\times$.

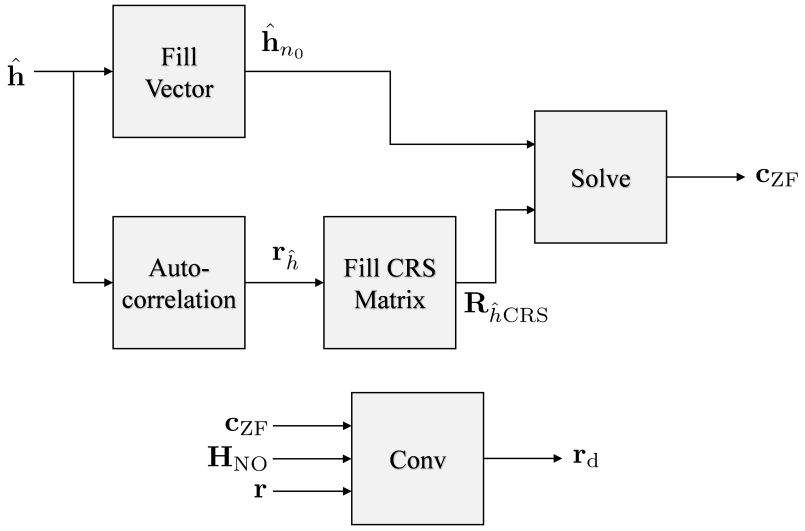



**Figure 8.3:** Block Diagram showing how the Zero-Forcing equalizer coefficients are implemented in the GPU.

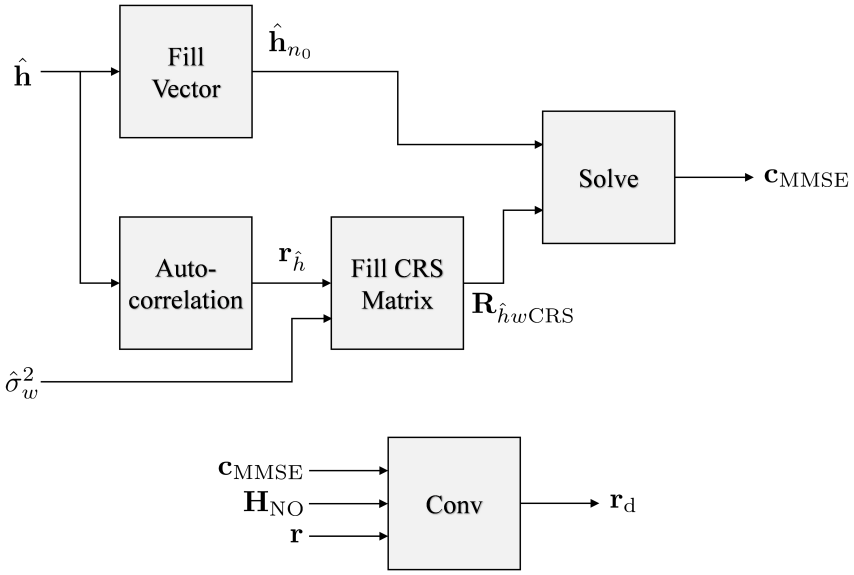



**Figure 8.4:** Block Diagram showing how the Minimum Mean Squared Error equalizer coefficients are implemented in the GPU.

**Table 8.1:** Defining start and stop lines for timing comparison in Listing 6.1.

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

## 8.2 Constant Modulus Algorithm GPU Implementation

The Constant Modulus Algorithm (CMA) is quite a bit more complicated than all other equalizers. CMA presented real challenges when implementing into a GPU. The direct approach didn't allow for multiple iterations. The more iterations CMA performed, the better CMA becomes becouse CMA is a steepest decent.

Each iteration of CMA uses the signal equalized by the most recent $\mathbf{c}_{\text{CMA}b}$

$$\mathbf{y} = \mathbf{r} * \mathbf{c}_{\text{CMA}b}. \tag{8.5}$$

The element of the gradient $\nabla J(k)$ is calculate by convolving the $12672$ sample $\mathbf{z}$ and with the $12672$ sample $\rho$. Note that all other frequency domain convolutions in this Thesis are Nfft $= 2^{14}$, but the convolution length $12672 + 12672 - 1 > 2^{14}$. The FFTs in the computation of $\nabla J(k)$ are Nfft $= 2^{15}$ point FFTs. The element of the gradient vector $\nabla J(k)$ is calculated by

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2 \tag{8.6}$$

where

$$\begin{aligned} b(n) &= \sum_{m=0}^{L_{pkt}-1} z(m)\rho(n-m) \\ &= \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-n) \end{aligned} \tag{8.7}$$

using convolution. Once $\nabla J$ is calculated, the steepest decent algorithm is applied

$$\mathbf{c}_{\text{CMA}(b+1)} = \mathbf{c}_{\text{CMA}(b)} - \mu \nabla J. \tag{8.8}$$

The goal of CMA is to perform as many iterations as possible. Once done iterating, apply the last $\mathbf{c}_{\text{CMA}(b+1)}$ to the received signal $\mathbf{r}$.

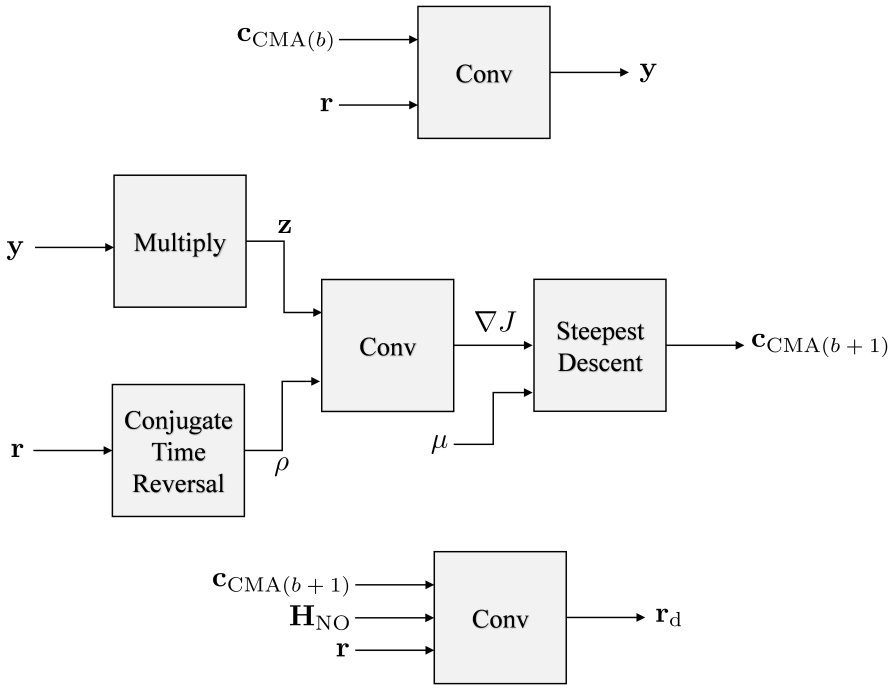



**Figure 8.5:** Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.

In Section 7.2, the direct application of CMA was massaged into convolution. Figure 8.5 shows a block diagram of how the CMA algorithm runs on the GPU.

The most computationally heavy part of CMA is computing $\nabla J(k)$. Computing $\nabla J(k)$ directly is almost $5\times$ slower than using convolution. The direct computation is done by performing a long $12672$ sample summation from Equation 7.28

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m) r^*(m-k), \quad -L_1 \leq k \leq L_2. \tag{8.9}$$

Long summations are slow in GPUs. Table 8.3 lists the comparison on computing $\nabla J(k)$ verse using convolution.

**Table 8.2:** The gradient vector $\nabla J(k)$ can be computed using convolution or computed directly.

| CMA Iteration Algorithm | Execution Time (ms) |
|---|---|
| $\nabla J$ directly | 421.317 |
| $\nabla J$ using convolution | 88.7743 |

**Table 8.3:** Defining start and stop lines for timing comparison in Listing 6.1.

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

## 8.3 Frequency Domain Equalizer One and Two GPU Implementation

The Frequency Domain Equalizers are by far the fastest and easiest to implement into GPUs. The block diagram looks just like convolution accept that point to point multiply isn't a simple two or three point complex multiply.

Equation 7.33 and is implemented directly in the GPU. To save execution time, the FFT of the detection filter $\mathbf{H}_{\text{NO}}$ multiplied at the same time FDE1 is calculated

$$R_{\text{d1}}(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})H_{\text{NO}}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \cdots, L-1 \tag{8.10}$$

$$R_{\text{d2}}(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})H_{\text{NO}}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \cdots, L-1 \tag{8.11}$$

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 $\omega_k$. Figures 8.6 and 8.7 show block diagrams of how FDE1 and FDE2 are implemented in the GPUs.

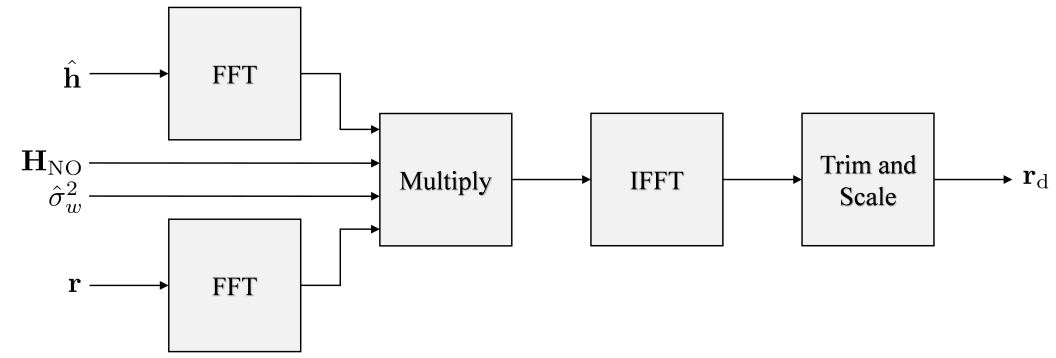



**Figure 8.6:** Diagram showing Frequency Domain Equalizer One is implemented in the frequency domain in GPUs.

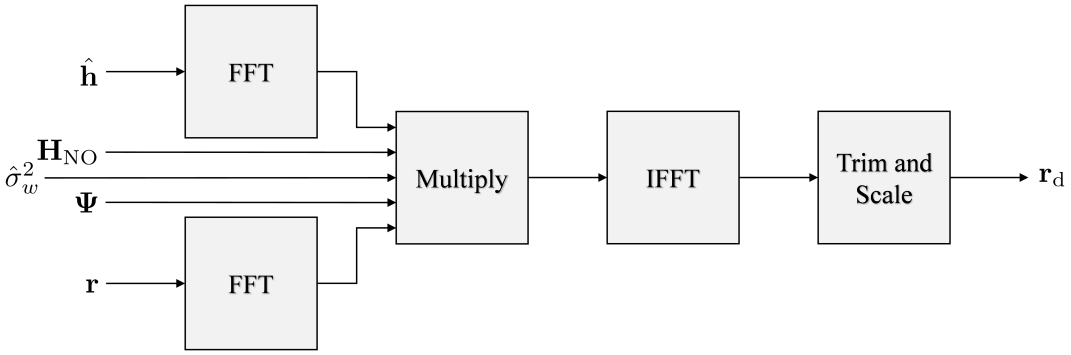



**Figure 8.7:** Diagram showing Frequency Domain Equalizer Two is implemented in the frequency domain in GPUs.

# Chapter 9

# Final Summary

this is the final summary

## Bibliography


[1] 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. 14

[2] 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. 16

[3] 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. 16

[4] E. Perrins, "FEC systems for aeronautical telemetry," **IEEE Transactions on Aerospace and Electronic Systems**, vol. 49, no. 4, pp. 2340–2352, October 2013. 17

[5] Wikipedia, "Graphics processing unit," 2015. [Online]. Available: http://en.wikipedia.org/wiki/Graphics_processing_unit 19

[6] NVIDIA, "Cuda toolkit documentation," 2017. [Online]. Available: http://docs.nvidia.com/cuda/ 19, 77

[7] 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. 44

[8] Wikipedia, "Fastest fourier transform in the west," 2017. [Online]. Available: http://www.fftw.org/ 36

[9] 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. 36

[10] 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. 64, 67, 69

[11] 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. 73

[12] Wikipedia, “Sparse matrix,” 2017. [Online]. Available: https://en.wikipedia.org/wiki/Sparse_matrix 77

[13] M. Hayes, **Statistical Digital Signal Processing and Modeling**. New York: John Wiley & Sons, 1996. 77