

GPU Implementation of Data-Aided Equalizers

Jeffrey T. Ravert

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

Master of Science

Michael D. Rice, Chair
Brian D. Jeffs
Brian A. Mazzeo

Department of Electrical and Computer Engineering

Brigham Young University

April 2017

Copyright © 2017 Jeffrey T. Ravert

All Rights Reserved

ABSTRACT

GPU Implementation of Data-Aided Equalizers

Jeffrey T. Ravert

Department of Electrical and Computer Engineering

Master of Science

Multipath is one of the dominant causes for link loss in aeronautical telemetry. Equalizers have been studied to combat multipath interference in aeronautical telemetry. Blind Constant Modulus Algorithm (CMA) equalizers are currently being used on SOQPSK-TG. The Preamble Assisted Equalization (PAQ) has been funded by the Air Force to study data-aided equalizers on SOQPSK-TG. PAQ compares side by side no equalization, data-aided zero forcing equalization, data-aided MMSE equalization, data-aided initialized CMA equalization, data-aided frequency domain equalization, and blind CMA equalization. A real time experimental test setup has been assembled including an RF receiver for data acquisition, FPGA for hardware interfacing and buffering, GPUs for signal processing, spectrum analyzer for viewing multipath events, and an 8 channel bit error rate tester to compare equalization performance. Lab tests were done with channel and noise emulators. Flight tests were conducted in March 2016 and June 2016 at Edwards Air Force Base to test the equalizers on live signals. The test setup achieved a 10Mbps throughput with a 6 second delay. 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
3.1 Overview	5
3.2 Preamble Detection	6
3.3 Frequency Offset Compensation	8
3.4 Channel Estimation	9
3.5 Noise Variance Estimation	9
3.6 Symbol-by-Symbol Detector	9
4 Signal Processing with GPUs	11
4.1 Simple GPU code example	11
4.2 GPU kernel using threads and thread blocks	14
4.3 GPU memory	15
4.4 Cuda Libraries	18
4.5 Thread Optimization	18

4.6	CPU GPU Pipelining	19
5	GPU Convolution	25
5.1	CUDA Batched Processing	25
5.2	Batched Convolution	26
5.3	Single Convolution	29
5.4	Batched Convolution	36
6	Equalizer Equations	55
6.1	Overview	55
6.2	Zero-Forcing and Minimum Mean Square Error Equalizers	55
6.2.1	Zero-Forcing	56
6.2.2	The Constant Modulus Algorithm	60
6.2.3	The Frequency Domain Equalizers	64
7	Equalizer GPU Implementation	67
7.1	Zero-Forcing and MMSE GPU Implementation	67
7.2	Constant Modulus Algorithm GPU Implementation	71
7.3	Frequency Domain Equalizer One and Two GPU Implementation	73
8	Final Summary	75
	Bibliography	76

List of Tables

4.1	The computational resources available with three NVIDIA GPUs used in this thesis (1x Tesla K40c 2x Tesla K20c).	16
5.1	Defining start and stop lines for timing comparison in Listing 5.1.	32
5.2	Convolution computation times with signal length 12672 and filter length 186 on a Tesla K40c GPU.	36
5.3	Convolution computation times with signal length 12672 and filter length 21 on a Tesla K40c GPU.	36
5.4	Defining start and stop lines for timing comparison in Listing 5.2.	38
5.5	Batched convolution execution times with for a 12672 sample signal and 186 tap filter on a Tesla K40c GPU.	41
5.6	Batched convolution execution times with for a 12672 sample signal and 21 tap filter on a Tesla K40c GPU.	41
5.7	Batched convolution execution times with for a 12672 sample signal and 206 tap filter on a Tesla K40c GPU.	41
5.8	Batched convolution execution times with for a 12672 sample signal and cascaded 21 and 186 tap filter on a Tesla K40c GPU.	42
7.1	Defining start and stop lines for timing comparison in Listing 5.1.	70
7.2	Defining start and stop lines for timing comparison in Listing 5.1.	73
7.3	Defining start and stop lines for timing comparison in Listing 5.1.	73

List of Figures

3.1	A block diagram of the estimation process.	5
3.2	A block diagram of the equalization and symbol detector process.	6
3.3	The iNET packet structure.	6
3.4	Offset Quadrature Phase Shift Keying symbol by symbol detector.	10
4.1	A block diagram of how a CPU sequentially performs vector addition.	12
4.2	A block diagram of how a GPU performs vector addition in parallel.	12
4.3	Block 0 32 threads launched in 4 thread blocks with 8 threads per block.	15
4.4	36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.	15
4.5	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.	16
4.6	NVIDIA Tesla K40c and K20c.	17
4.7	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.	17
4.8	The GPU convolution thread optimization of a 12672 length signal with a 186 tap filter using shared memory. 192 is the optimal number of threads per block executing in 0.1101ms. Note that at least 186 threads per block must be launched to compute correct output.	20
4.9	ConvGPU thread optimization 128 threads per block 0.006811.	21
4.10	The typical approach of CPU and GPU operations. This block diagram shows a Profile of Listing 4.2.	21
4.11	GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 4.3.	22

4.12 A block diagram of pipelining a CPU with three GPUs.	23
5.1 Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	25
5.2 Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	26
5.3 Comparison of number of floating point operations (flops) required to convolve a 12672 sample complex signal with a 186 tap complex filter.	30
5.4 Comparison of number of floating point operations (flops) required to convolve a 12672 sample complex signal with a 21 tap complex filter.	31
5.5 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.	32
5.6 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.	33
5.7 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.	34
5.8 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.	35
5.9 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.	37
5.10 Comparison of a batched complex convolution on a GPU. The signal length is varied and the filter is fixed at 186 taps.	38
5.11 Comparison of a batched complex convolution on a GPU. The signal length is varied and the filter is fixed at 21 taps.	39
5.12 Comparison of a batched complex convolution on a GPU. The filter length is varied and the signal length is set at 12672 samples.	40
5.13 Two ways to convolve the signal r with the 186 tap filter c and 21 tap filter d	42
6.1 Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	63
6.2 I need help on this one!!!!	65

7.1	Convolution of vectors \mathbf{c} and \mathbf{r} block diagram simplified to one block marked Conv.	68
7.2	Convolution of vectors \mathbf{c} , \mathbf{r} and \mathbf{H}_{NO} block diagram simplified to one block marked Conv.	68
7.3	Block Diagram showing how the Zero-Forcing equalizer coefficients are implemented in the GPU.	70
7.4	Block Diagram showing how the Minimum Mean Squared Error equalizer coefficients are implemented in the GPU.	70
7.5	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.	72
7.6	Diagram showing Frequency Domain Equalizer One is implemented in the frequency domain in GPUs.	74
7.7	Diagram showing Frequency Domain Equalizer Two is implemented in the frequency domain in GPUs.	74

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 achieve 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

3.1 Overview

This chapter gives a high lever overview of the algorithms required to implement equalizers in GPUs. As shown in chapter blah, an equalizer uses the channel estimate and the noise varinace to be calculated. The channel estimate is found by estimating the channel based on the preamble. The noise estimate is found by estimating the noise varinace from the preamble after the channel has been estimated. The frequency offset also needs to be removed before estimating the channel and noise. All estimators are data aided. The preamble has to be found before any estimators can be ran.

A block diagram of the estimators is shown in Figure 3.1. A block diagram of the equalization and symbol detector is shown in Figure 3.1. The estimators and symbol detector will be explained briefly in this chapter. The equalizer computation and detection filter process in the dashed box will be explained in chapter 6.

Figure 3.1: A block diagram of the estimation process.

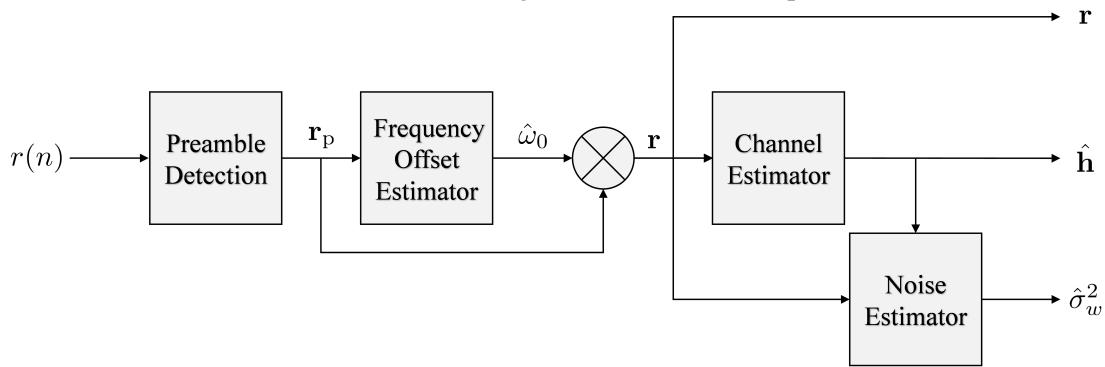


Figure 3.2: A block diagram of the equalization and symbol detector process.

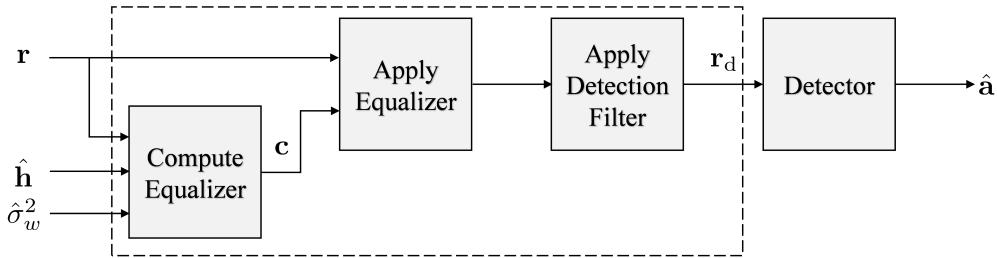
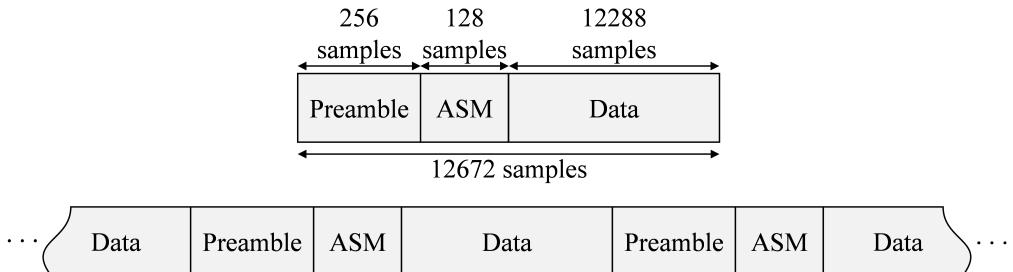


Figure 3.3: The iNET packet structure.



3.2 Preamble Detection

The received samples in this thesis has the iNET packet structure shown in Figure 3.3. 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 CD98_{hex} and the ASM field

$$034776C72728950B0_{\text{hex}} \quad (3.1)$$

Each 16-bit sequence CD98_{hex} sampled at two samples/bit are $L_q = 32$ samples long.

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_{pkt} vectors with the structure shown in Figure 3.3. Each vector contains $L_p = 256$ preamble samples, $L_{\text{ASM}} = 136$ ASM samples and $L_d = 12288$ data samples. The full length of a vector is $L_p + L_{\text{ASM}} + L_d = 12672$.

Before the structuring the received samples into packets, the preambles are found using a preamble detector explained in [1]. Equations (3.2) through (3.5) have been optimized for GPUs and are implemented directly.

$$L(u) = \sum_{m=0}^7 [I^2(n, m) + Q^2(n, m)] \quad (3.2)$$

where the inner summations are

$$\begin{aligned} I(n, m) \approx & \sum_{\ell \in \mathcal{L}_1} r_R(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_2} r_R(\ell + 32m + n) + \sum_{\ell \in \mathcal{L}_3} r_I(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_4} r_I(\ell + 32m + n) \\ & + 0.7071 \left[\sum_{\ell \in \mathcal{L}_5} r_R(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_6} r_R(\ell + 32m + n) \right. \\ & \quad \left. + \sum_{\ell \in \mathcal{L}_7} r_I(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_8} r_I(\ell + 32m + n) \right], \quad (3.3) \end{aligned}$$

and

$$\begin{aligned} Q(n, m) \approx & \sum_{\ell \in \mathcal{L}_1} r_I(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_2} r_I(\ell + 32m + n) \\ & - \sum_{\ell \in \mathcal{L}_3} r_R(\ell + 32m + n) + \sum_{\ell \in \mathcal{L}_4} r_R(\ell + 32m + n) \\ & + 0.7071 \left[\sum_{\ell \in \mathcal{L}_5} r_I(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_6} r_I(\ell + 32m + n) \right. \\ & \quad \left. - \sum_{\ell \in \mathcal{L}_7} r_R(\ell + 32m + n) + \sum_{\ell \in \mathcal{L}_8} r_R(\ell + 32m + n) \right] \quad (3.4) \end{aligned}$$

with

$$\begin{aligned}
\mathcal{L}_1 &= \{0, 8, 16, 24\} \\
\mathcal{L}_2 &= \{4, 20\} \\
\mathcal{L}_3 &= \{2, 10, 14, 22\} \\
\mathcal{L}_4 &= \{6, 18, 26, 30\} \\
\mathcal{L}_5 &= \{1, 7, 9, 15, 17, 23, 25, 31\} \\
\mathcal{L}_6 &= \{3, 5, 11, 12, 13, 19, 21, 27, 28, 29\} \\
\mathcal{L}_7 &= \{1, 3, 9, 11, 12, 13, 15, 21, 23\} \\
\mathcal{L}_8 &= \{5, 7, 17, 19, 25, 27, 28, 29, 31\}.
\end{aligned} \tag{3.5}$$

A correlation peak in $L(u)$ indicate the starting index k of a preamble. The vector \mathbf{r}_p in Figure 3.1 is defined by

$$\mathbf{r}_p = \begin{bmatrix} r(k) \\ \vdots \\ r(k + L_{\text{pkt}} - 1) \end{bmatrix} = \begin{bmatrix} r_p(0) \\ \vdots \\ r_p(L_{\text{pkt}} - 1) \end{bmatrix} \tag{3.6}$$

3.3 Frequency Offset Compensation

The frequency offset estimator shown in Figure 3.1 is the estimator taken from [2, eq. (24)]. With the notation adjusted slightly, the frequency offset estimate is

$$\hat{\omega}_0 = \frac{1}{L_q} \arg \left\{ \sum_{n=i+2L_q}^{i+7L_q-1} r_p(n)r_p^*(n-L_q) \right\} \quad \text{for } i = 1, 2, 3, 4, 5. \tag{3.7}$$

The frequency offset is estimated for every packet or each vector \mathbf{r}_p .

The frequency offset is compensated for by derotating the packet structured samples by the estimated offset

$$r(n) = r_p(n)e^{-j\hat{\omega}_0}. \tag{3.8}$$

Equations (3.7) and (3.8) are easily implemented into GPUs.

3.4 Channel Estimation

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

$$\hat{\mathbf{h}} = \underbrace{(\mathbf{X}^\dagger \mathbf{X})^{-1} \mathbf{X}^\dagger}_{\mathbf{X}_{\text{lpi}}} \mathbf{r} \quad (3.9)$$

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

$$\hat{\mathbf{h}} = \mathbf{X}_{\text{lpi}} \mathbf{r}. \quad (3.10)$$

The matrix operation $\mathbf{X}_{\text{lpi}} \mathbf{r}$ maps simply and efficiently in GPUs using cuBLAS.

3.5 Noise Variance Estimation

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

$$\hat{\sigma}_w^2 = \frac{1}{2\rho} \left| \mathbf{r} - \mathbf{X} \hat{\mathbf{h}} \right|^2 \quad (3.11)$$

where

$$\rho = \text{Trace} \left\{ \mathbf{I} - \mathbf{X} (\mathbf{X}^\dagger \mathbf{X})^{-1} \mathbf{X}^\dagger \right\}. \quad (3.12)$$

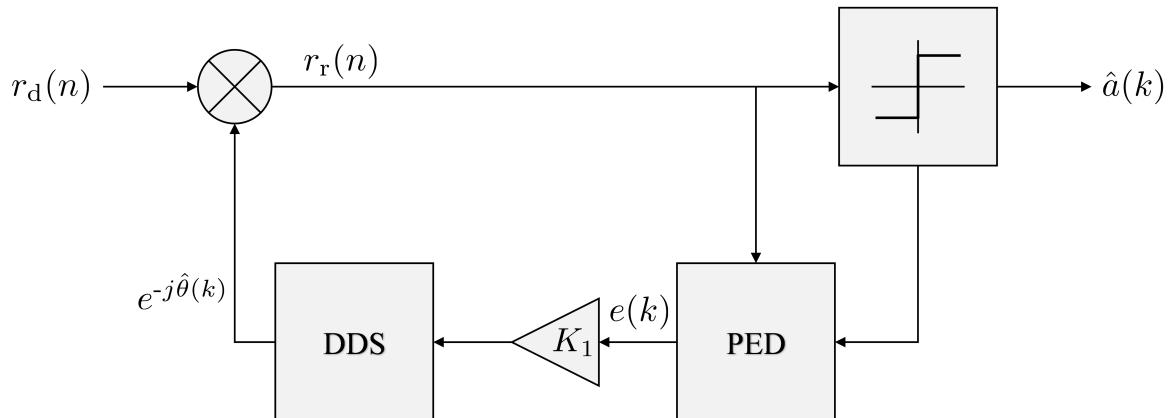
Equation (3.11) is easily implemented into GPUs.

3.6 Symbol-by-Symbol Detector

The symbol by symbol detector block in Figure 3.2 is a Offset Quadrature Phase Shift Keying (OQPSK) detector. Using the simple OQPSK detector in place of the complex MLSE SOQPSK-TG detector leads to less than 1dB in bit error rate [4].

A Phase Lock Loop (PLL) is needed in the SxS 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, a PLL is required to eliminate residual frequency offset. The PLL tracks out the residual frequency offset using a feedback control loop.

Figure 3.4: Offset Quadrature Phase Shift Keying symbol by symbol detector.



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

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

Implementing the PLL with a feedback loop seems like it may be challenging in GPUs because it cannot be parallelized. While the PLL cannot be parallelized on a sample by sample basis, it can be parallelized on a packet by packet basis. Running the PLL and detector serially through a full packet of data is still relatively fast because each iteration of the PLL and detector is computationally light.

Chapter 4

Signal Processing with GPUs

This thesis explores the use of GPUs in data-aided estimation, equalization and filtering operations.

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.

The purpose of this overview is to provide context for the contributions of this thesis. As such this overview is not a tutorial. For a full explanation of CUDA programming please see the CUDA toolkit documentation [6].

4.1 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

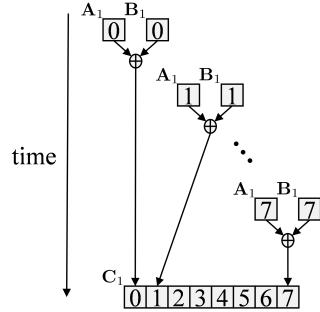


Figure 4.1: A block diagram of how a CPU sequentially performs vector addition.

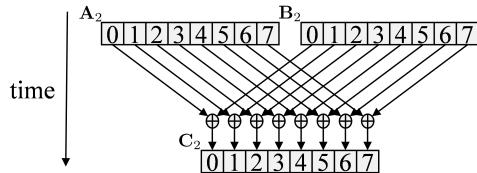


Figure 4.2: A block diagram of how a GPU performs vector addition in parallel.

kernels have finished, the resulting memory has to be copied back from the device (GPU) to the host (CPU).

Listing 4.1 shows a simple program that sums two vectors together

$$\begin{aligned} \mathbf{C}_1 &= \mathbf{A}_1 + \mathbf{B}_1 \\ \mathbf{C}_2 &= \mathbf{A}_2 + \mathbf{B}_2 \end{aligned} \tag{4.1}$$

where each vector is length 1024. Figure 4.1 shows how the CPU computes \mathbf{C}_1 by summing elements of \mathbf{A}_1 and \mathbf{B}_1 together *sequentially*. Figure 4.2 shows how the GPU computes \mathbf{C}_2 by summing elements of \mathbf{A}_2 and \mathbf{B}_2 together *in parallel*. The GPU kernel computes every element of \mathbf{C}_2 in parallel while the CPU computes one element of \mathbf{C}_1 at a time.

Listing 4.1: Comparison of CPU verse GPU code.

```

1 #include <iostream>
2 #include <stdlib.h>
3 #include <math.h>
4 using namespace std;
5
6 void VecAddCPU(float* destination, float* source0, float* source1, int myLength) {
7     for(int i = 0; i < myLength; i++)
8         destination[i] = source0[i] + source1[i];
9 }
10
11 __global__ void VecAddGPU(float* destination, float* source0, float* source1, int lastThread) {
12     int i = blockIdx.x*blockDim.x + threadIdx.x;
13
14     // don't access elements out of bounds
15     if(i >= lastThread)
16         return;
17
18     destination[i] = source0[i] + source1[i];
19 }
20
21
22 int main(){
23     int numPoints = pow(2,22);
24     cout << numPoints << endl;
25     /*****
26                     CPU Start
27     *****/
28     // allocate memory on host
29     float *A1;
30     float *B1;
31     float *C1;
32     A1 = (float*) malloc (numPoints*sizeof(float));
33     B1 = (float*) malloc (numPoints*sizeof(float));
34     C1 = (float*) malloc (numPoints*sizeof(float));
35
36     // Initialize vectors 0-99
37     for(int i = 0; i < numPoints; i++){
38         A1[i] = rand()%100;
39         B1[i] = rand()%100;
40     }
41
42     // vector sum C1 = A1 + B1
43     VecAddCPU(C1, A1, B1, numPoints);
44     /*****
45                     CPU End
46     *****/
47
48     /*****
49                     GPU End
50     *****/
51     // allocate memory on host for result
52     float *C2;
53     C2 = (float*) malloc (numPoints*sizeof(float));
54
55     // allocate memory on device for computation
56     float *A2_gpu;
57     float *B2_gpu;
58     float *C2_gpu;
59     cudaMalloc(&A2_gpu, sizeof(float)*numPoints);
60     cudaMalloc(&B2_gpu, sizeof(float)*numPoints);
61     cudaMalloc(&C2_gpu, sizeof(float)*numPoints);
62
63     // Copy vectors A and B from host to device
64     cudaMemcpy(A2_gpu, A1, sizeof(float)*numPoints, cudaMemcpyHostToDevice);
65     cudaMemcpy(B2_gpu, B1, sizeof(float)*numPoints, cudaMemcpyHostToDevice);
66 }
```

```

67     // Set optimal number of threads per block
68     int numThreadsPerBlock = 32;
69
70     // Compute number of blocks for set number of threads
71     int numBlocks = numPoints/numThreadsPerBlock;
72
73     // If there are left over points, run an extra block
74     if(numPoints % numThreadsPerBlock > 0)
75         numBlocks++;
76
77     // Run computation on device
78     //for(int i = 0; i < 100; i++)
79     VecAddGPU<<<numBlocks, numThreadsPerBlock>>>(C2_gpu, A2_gpu, B2_gpu, numPoints);
80
81     // Copy vector C2 from device to host
82     cudaMemcpy(C2, C2_gpu, sizeof(float)*numPoints, cudaMemcpyDeviceToHost);
83     /*-----*
84                                     GPU End
85     -----*/
86
87     // Compare C2 to C1
88     bool equal = true;
89     for(int i = 0; i < numPoints; i++)
90         if(C1[i] != C2[i])
91             equal = false;
92     if(equal)
93         cout << "C2 is equal to C1." << endl;
94     else
95         cout << "C2 is NOT equal to C1." << endl;
96     sleep(2);
97
98     // Free vectors on CPU
99     free(A1);
100    free(B1);
101    free(C1);
102    free(C2);
103
104    // Free vectors on GPU
105    cudaFree(A2_gpu);
106    cudaFree(B2_gpu);
107    cudaFree(C2_gpu);
108 }

```

4.2 GPU kernel using threads and thread blocks

A GPU kernel is executed on a GPU by launching $\text{numThreadsPerBlock} \times \text{numBlocks}$ threads. Each thread has a unique index. CUDA calls this index `threadIdx` and `blockIdx`. `threadIdx` is the thread index inside the assigned thread block. `blockIdx` is the index of the block the thread is assigned. `blockDim` is the number of threads assigned per block, in fact `blockDim = numThreadsPerBlock`. 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 vectors.

To replace a CPU for loop that runs 0 to $N - 1$, a GPU kernel launches N threads 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

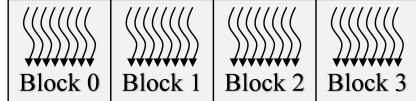


Figure 4.3: Block 0 32 threads launched in 4 thread blocks with 8 threads per block.

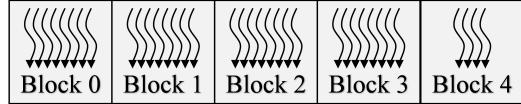


Figure 4.4: 36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.

integer multiple of T . Figure 4.3 shows 32 threads launched in 4 thread blocks with 8 threads per block. Figure 4.4 shows 36 threads launched in 5 thread blocks with 8 threads per block. An full extra thread block must be launched to with 8 threads but 4 threads are idle.

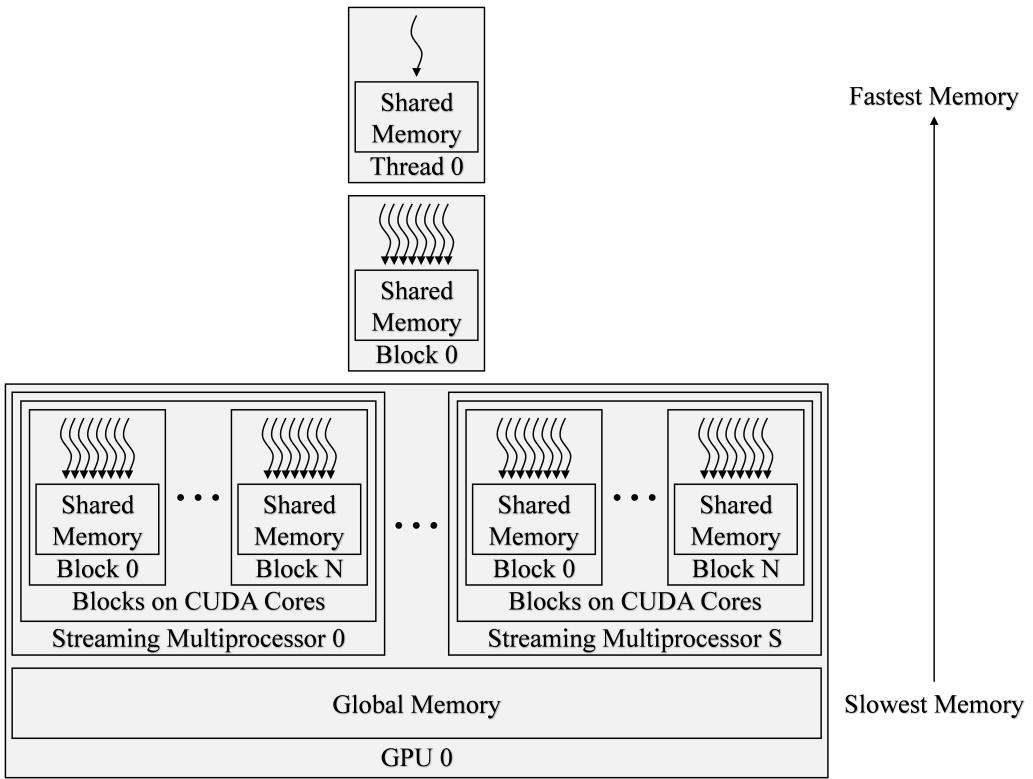
4.3 GPU memory

Thread blocks run independent of other thread blocks. The GPU does not guarantee Block 0 will execute before Block 2. Threads in 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 private shared memory in 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 is a few clock cycles. Why not just do all computations in local memory? The memory needs come from global memory to before it can be used in local memory. Memory should be saved in shared memory if many threads are going to use it in a thread block. 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 will shed some light. This thesis uses NVIDIA Tesla K40c and K20c GPUs, Table 4.1 gives some specifications and Figure 4.6 shows the form factor of the these GPUs. The red box in Figure 4.7 show the GPU

Figure 4.5: 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.



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

Table 4.1: The computational resources available with three NVIDIA GPUs used in this thesis (1x Tesla K40c 2x Tesla K20c).

chip and the yellow boxes show the SRAM that is *off* the GPU chip. The GPU 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. Now 400-800 clock cycles doesn't sound all the bad huh?



Figure 4.6: NVIDIA Tesla K40c and K20c.

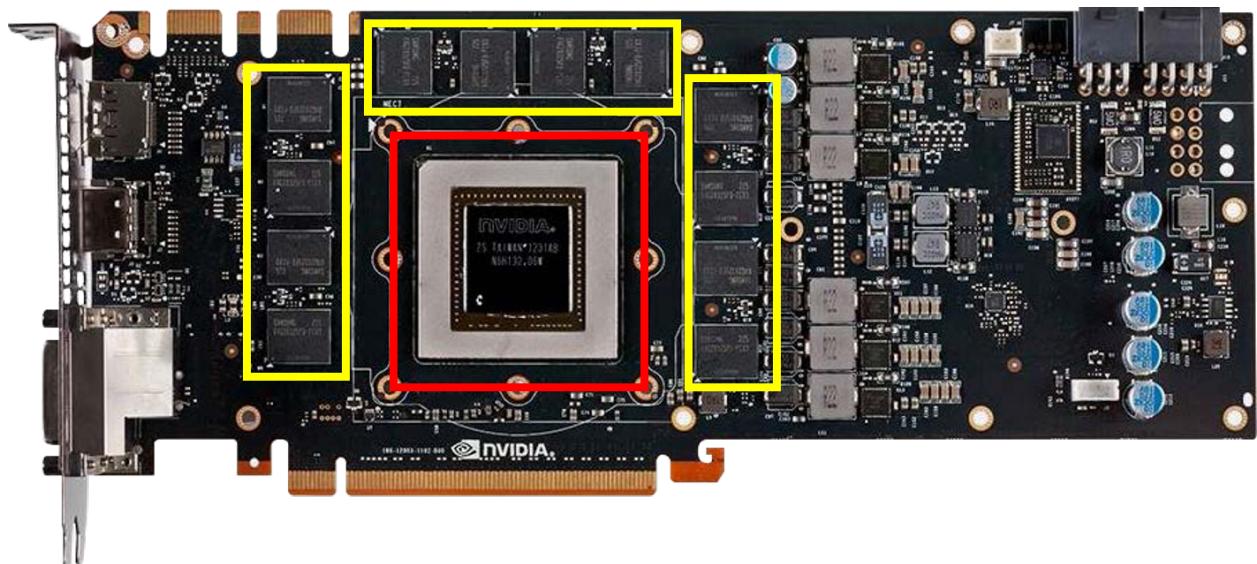


Figure 4.7: Example of an NVIDIA GPU card. The SRAM is shown to be boxed in yellow. The GPU chip is shown to be boxed in red.

4.4 Cuda Libraries

CUDA isn't just a programming language, it also has many GPU optimized libraries that are extremely useful. CUDA libraries are written by NVIDIA engineers that know how to squeeze out every drop of performance out of NVIDIA GPUs. Because ninjas are unbeatable, NVIDIA engineers are known as ninjas in the Telemetry Group at BYU. While figuring out how to optimize a GPU kernel is extremely satisfying, GPU programmers should always search the CUDA library documentation for any thing that might be useful.

Some libraries used in this Thesis are

- cufft
- cublas
- cusolver
- cusolverSp

4.5 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 block. Running the GPU at high occupancy provides each thread with less resources.

Launching 1024 threads per block isn't always bad though. 1024 might be the optimal number of threads per block for a given GPU kernel. 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. A simple test program could time a GPU kernel while sweeping the number of threads per block from 1 to 1024. The number of threads per block with the fastest computation time is the optimal number of threads per block for that specific GPU kernel.

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 4.8 shows the execution time of ConvGPUshared while varying threads per block. Although the minimum execution time is 0.1078ms at the optimal 96 threads per block, incorrect output will result if ConvGPUshared is launched with less than 186 threads per block. Launching 96 threads per block only transfer 96 filter coefficients to shared memory from global memory. Luckily, launching 192 threads per block is near optimal with an execution time of 0.1101ms. By simply adjusting the number of threads per block, ConvGPUshared can have a 2 \times speed up.

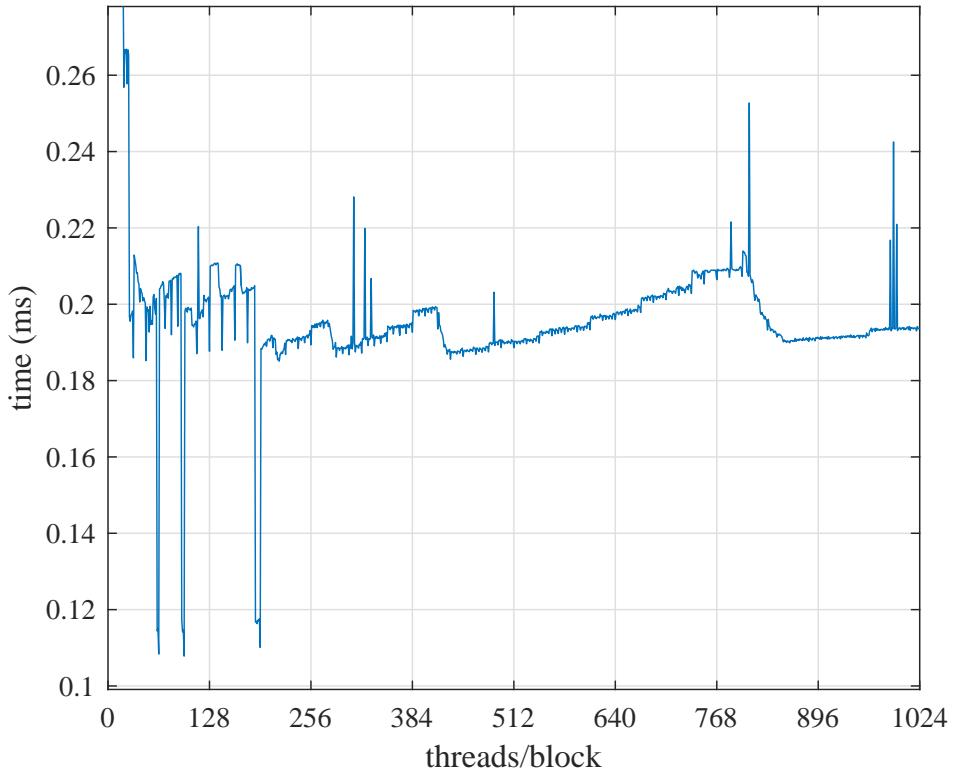
Adjusting the number of threads per block doesn't always drastically speed up GPU kernels. Figure 4.8 shows the execution time for ConvGPU with varying threads per block. Launching 560 does produce about a 1.12x speed up, but thread optimization doesn't have as much of an affect of ConvGPU verse ConvGPUshared.

To answer the question: "There are X number of ways to implement this algorithm, which one is executes the fastest?" the answer always is "It depends. Implement the algorithm X ways and see which is fastest."

4.6 CPU GPU Pipelining

Typically a programmer acquires data, crunches the data then he is done. But what if you could crunch data while acquiring data? How much computation time could you gain by pipelining acquiring data and processing data?

Figure 4.8: The GPU convolution thread optimization of a 12672 length signal with a 186 tap filter using shared memory. 192 is the optimal number of threads per block executing in 0.1101ms. Note that at least 186 threads per block must be launched to compute correct output.



Listing 4.2 shows example code of a straight forward structure of a typical implementation. The CPU acquires data from myADC on Line 5. After taking time to acquire data, the CPU launches GPU instructions on lines 8-10. cudaDeviceSynchronize on line 13 blocks the CPU until all instructions are done on the GPU.

Figure 4.10 shows a block diagram of what is happening on the CPU and GPU in Listing 4.2. 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 4.9: ConvGPU thread optimization 128 threads per block 0.006811.

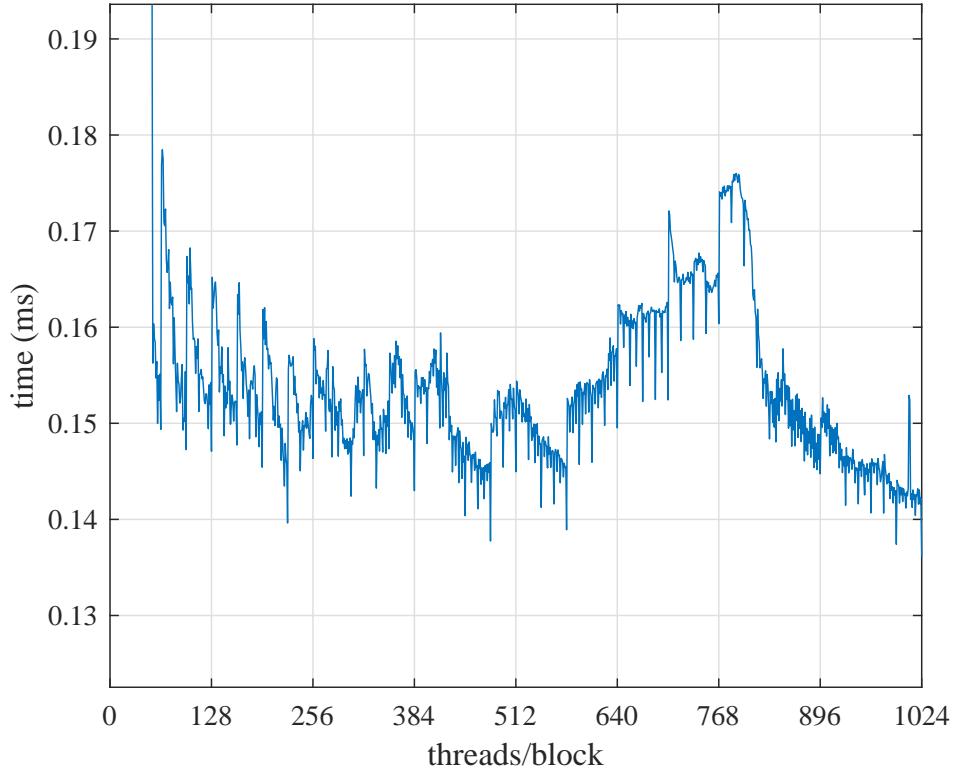


Figure 4.10: The typical approach of CPU and GPU operations. This block diagram shows a Profile of Listing 4.2.

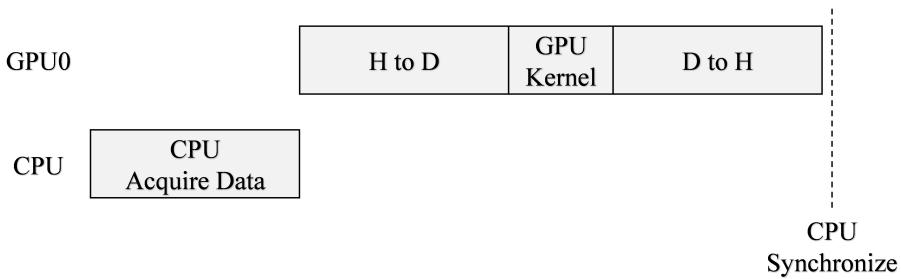
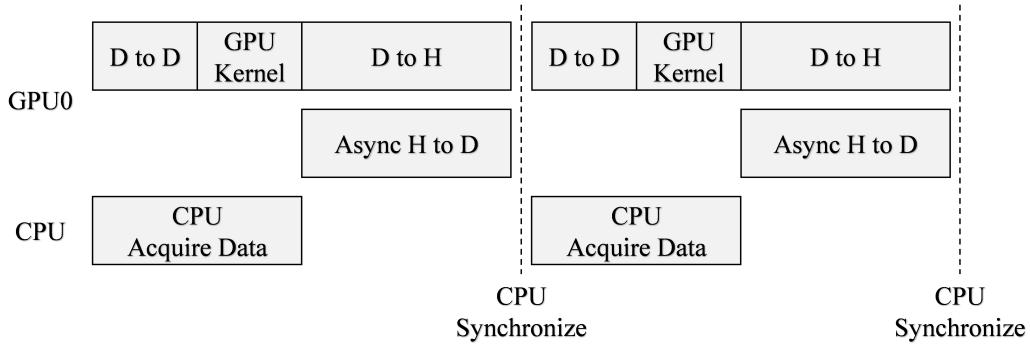


Figure 4.11: GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 4.3.



Listing 4.2: 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 Acquire 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 }
```

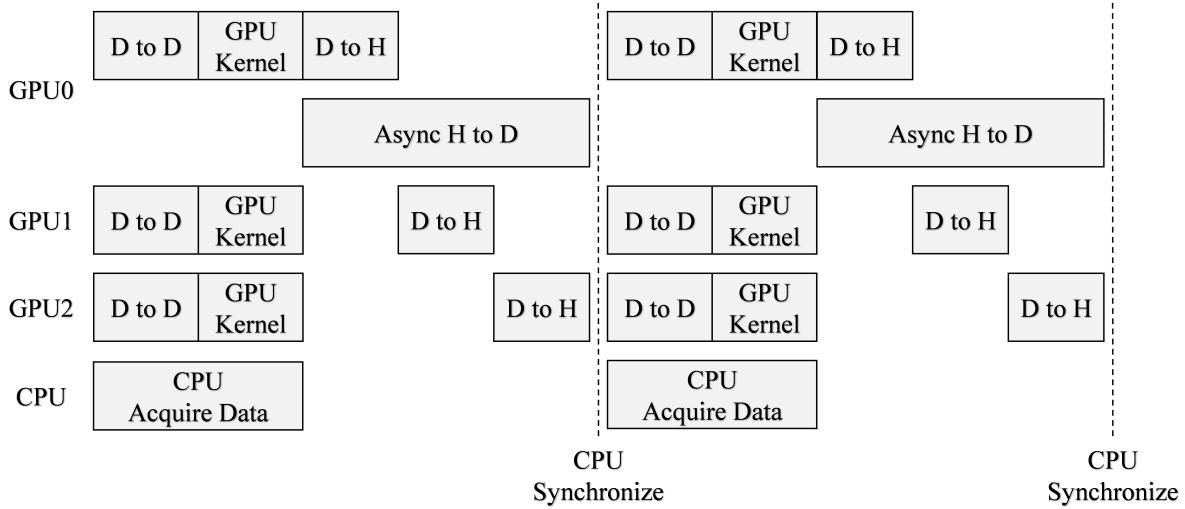
So the question is, “Can the throughput increase by using idle time on the GPU and CPU?”

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 a CPU than the GPU. Once the CPU has finished its operations, the CPU calls to wait for the GPU to finish.

Listing 4.3 shows how to pipeline CPU and GPU operations. Instead of acquiring data first, 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 latency equal a full cycle of data.

Listing 4.3: 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.

Figure 4.12: A block diagram of pipelining a CPU with three GPUs.



```

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

Pipelineing can be extended to multiple GPUs for even more throughput but only suffer latency of one GPU. Figure 4.12 shows a block diagram of how three GPUs can be pipelined. A strong understanding of your full system is required to pipeline at this level.

Chapter 5

GPU Convolution

5.1 CUDA Batched Processing

Typically when batched processing, each batch is ran independent of other batches. In CPU this is done by calling the same function on different data. Shouldn't GPUs be able to do the same thing...but in parallel.

CUDA has many libraries that are “batched,” meaning a GPU kernel launched for each batch of data. Examples of using a batched libraries that are used in PAQ are cuFFT, cuBLAS and cuSolver. Each of these libraries have batched kernels. Figure 5.1 shows the concept of a batched matrix multiply.

Batched libraries perform much better than calling a GPU kernel multiple times with independent data. Haidar et al. showed batched libraries in GPUs achieve more Gflops per second than calling GPU kernels multiple times [7]. Batched processing performs very well in GPUs because

Figure 5.1: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.

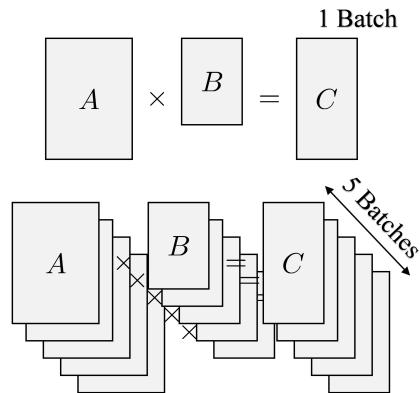
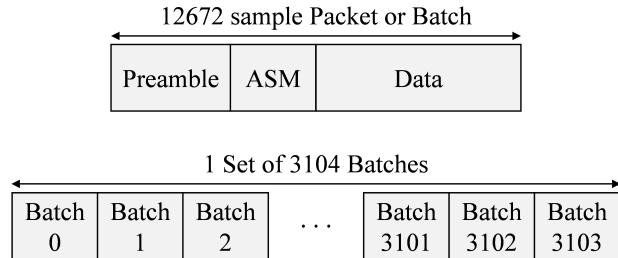


Figure 5.2: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.



it increases parallelism, gives more opportunities for NVIDIA engineers to optimize and reduces overhead on the CPU. The bottom line is use batched libraries as often as possible.

The PAQ system is suited very well for batched processing. Each 1.907 second set has 3104 packets or batches of data as shown in Figure 5.2. If an operation is done on each packet, available CUDA batched libraries should always be used. For simplicity, every block diagram shown in this chapter only applies to a single batch. Every block diagram is batched and applied to 3104 iNET packets.

5.2 Batched Convolution

Chapter 4 delved into comparing a single convolution in CPUs and GPUs in the time domain or frequency domain. As talked about in Chapter blah, system overview, We dont only have a packet or batch, we have 3104 packets or batches. When we perform convolution we convolve 3104 signals with 3104 filters. So... How does the problem change? Obviously doing batched convolution in a CPU will not be feasible and wont be considered.

Lets stay in GPU land. Should we do convolution in the time domain or the frequency domain? Are there draw backs to staying in the time domain? If we do convolution in the time domain, should we use shared memory? Are there draw backs for going to the frequency domain?

Get timing of convolution in the time domain global, time domain shared, time domain frequency.

There is one very important thing we have over looked... We are doing just one convolution...we are doing two. We have to convolve with the equalizer AND the numerically optimized detection filter \mathbf{h}_{NO} . If we stay in the time domain, we have to do two cascaded convolutions. Two convolutions takes...twice the time.... If we go to the frequency domain, the point to point or hadamard product is done with 3 inputs rather than 2. Convolving with 2 inputs vs even 10 inputs is about the same. The only required part is performing the FFT on the inputs and performing the N way multiply...

Well, \mathbf{h}_{NO} doesn't change, so we compute the FFT of the detection filter \mathbf{H}_{NO} and store it. So doing the convolution requires taking the FFT of 2 vectors \mathbf{c} and \mathbf{r} . After the hadamard product, the functions independent of how many filters we are convolving with accept for we may have to trip off different parts depending on what we are convolving with.

Optimizing convolution speeds up every equalizer because every equalizer uses convolution...CMA uses convolution twice per iteration.

Chapter 6 the equalizer equations were conditioned for GPU implementation. This Chapter explains how the equalizers were implemented into GPUs. Block diagrams will be a very helpful tool to display how the GPU kernels flow.

CUDA has many functions that are “batched,” meaning the GPU can apply the same function to each packet. Don’t be confused with the term batch. In the PAQ system, a “batch” is 1.907 seconds of data or 3103 or 3104 packets. In CUDA, “batched” kernels are GPU kernels that are called 3103 or 3104 times to run on each iNET packet. One could say “The batched kernel runs on a batch of data by processing 3103 or 3104 packets.” The CUDA number of batches is how many iNET packets are being processed.

This sections explains how each equalizer is computed and how the “numerically optimized” H_{NO} detection filter is applied in each case [4, Fig. 3].

Convolution is one of the most important tools in any digital signal processing engineer’s toolbox. In fact, Chapter blah (system overview) shows that convolution is used nearly 10 times.

Much time was spent optimizing convolution because even a 10 ms speed up provides 100ms of extra processing time.

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

$$y(n) = \sum_{m=0}^{L-1} x(m)h(n-m) \quad (5.1)$$

and discrete time convolution computed in the frequency domain is

$$\mathbf{y} = \mathcal{F}^{-1}(\mathcal{F}(\mathbf{x}) \times \mathcal{F}(\mathbf{h})) \quad (5.2)$$

where the N sample complex signal is convolved with the complex L tap filter.

Theory says if the given filter is relatively long the frequency domain is the best choice. What constitutes a “long” signal? When should convolution be done in the frequency domain rather than time domain?

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

$$(A + jB) \times (C + jD) = (AC - BD) + j(AD + BC) \quad (5.3)$$

is 6 flops, 4 multiplies and 2 additions/subtractions. Each output element of \mathbf{y} in Equation (5.1) requires $8L$ flops, 2 flops (real and imaginary) for each term to be summed then 6 flops for time signal and filter complex multiply. The length of the output \mathbf{y} is $N + L - 1$. The number of flops required for convolution is

$$8L(N + L - 1) \text{ flops.} \quad (5.4)$$

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 zero pad out to the next power of to above M . The current most popular FFT library 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 performs $5M \log_2(M)$ flops [8,9]. Performing convolution in the frequency domain requires

$$3 \times 5M \log_2(M) + 6M \text{ flops.} \quad (5.5)$$

5.3 Single Convolution

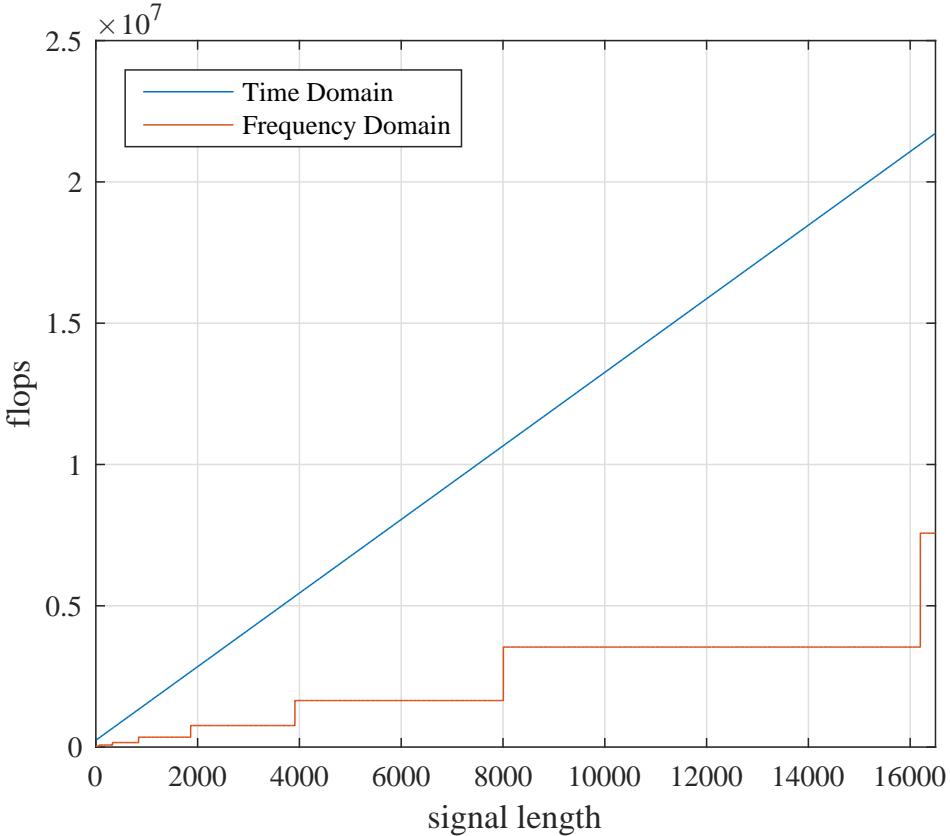
Figure 5.3 compares the number of flops required for time domain verse frequency domain convolution of a 12672 sample complex signal with a 186 tap complex filter. Figure 5.4 compares the number of flops required for time domain verse frequency domain convolution of a 12672 sample complex signal with a 21 tap complex filter. Appending zeros to the next power of 2 causes the stair stepping pattern. Judging by the figures, a 186 tap filter is long and a 21 tap filter is short compared to the 12672 sample signal.

Does the number of flops affect CPU or GPU execution time? Listing 5.1 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
- frequency domain convolution in a GPU using CUDA libraries

The CPU implements Equation (5.1) in ConvCPU directly on line 209 using a function from lines 11 to 34. The CPU implements Equation (5.2) 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 implements time domain convolution by accessing global memory for every element of the signal and filter. The

Figure 5.3: Comparison of number of floating point operations (flops) required to convolve a 12672 sample complex signal with a 186 tap complex filter.

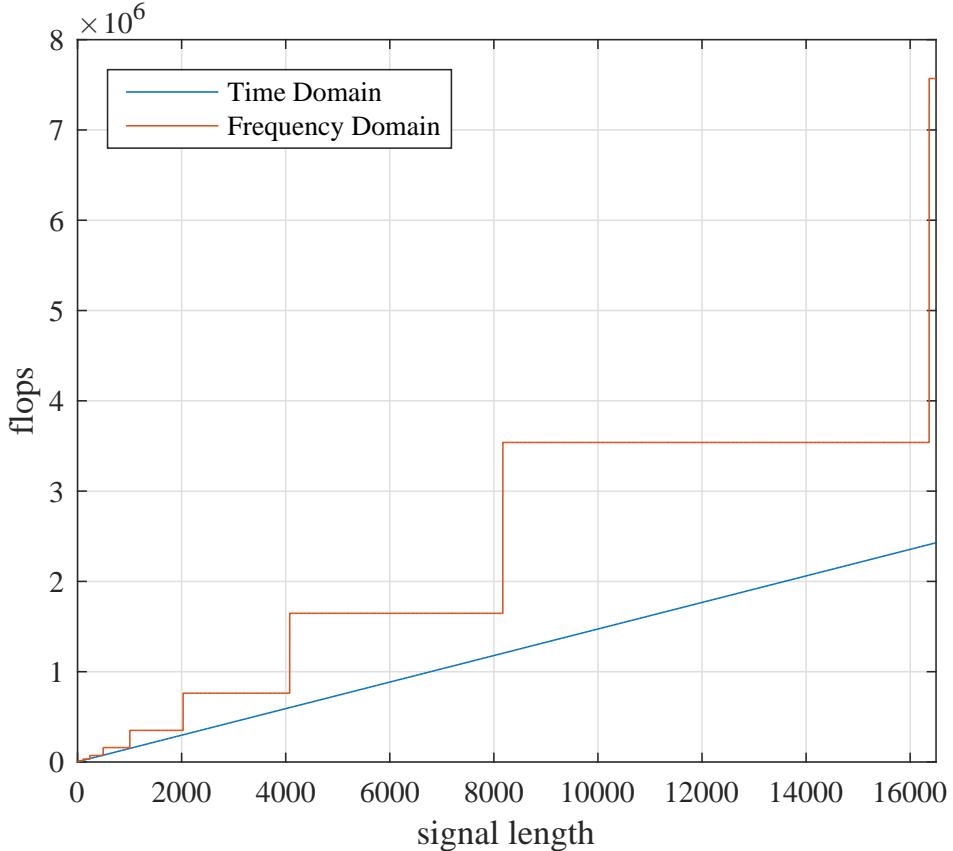


GPU implements time domain convolution using shared memory in lines 283 to 292. The GPU kernel ConvGPUshared on lines 67 to 101 is nearly identical to ConvGPU. Threads accessing the same elements of the filter in global memory is a waste of valuable clock cycles. ConvGPUshared pays and initial price on lines 72 to 76 to move L_h filter coefficients from the off chip 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 it worth the initial cost to used shared memory? When should convolution be done in the frequency domain?

The short answer to all of the questions is it depends on your signal length, filter length, CPU, GPU and memory. A good CUDA programmer can make an educated guess on which

Figure 5.4: Comparison of number of floating point operations (flops) required to convolve a 12672 sample complex signal with a 21 tap complex filter.



algorithm may be faster, 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 4.1 was timed. All the memory transfers to and from the host were timed for a fair comparison of GPU to CPU. Table 5.1 shows where timing was started and stopped for each algorithm.

Figure 5.5 compares the computation time of a fixed length 186 tap filter convolved with a varied length signal. The execution time varies enough that the plot is messy and unreadable. Figure 5.6 compares the same data but 15 sample local minimums were found.

With the plot lower bounded, compare Figure 5.6 to Figure 5.3. Does the CPU and GPU follow the same trend as the number of flops? The CPU has the exact structure that the number

Table 5.1: Defining start and stop lines for timing comparison in Listing 5.1.

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

Figure 5.5: 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.

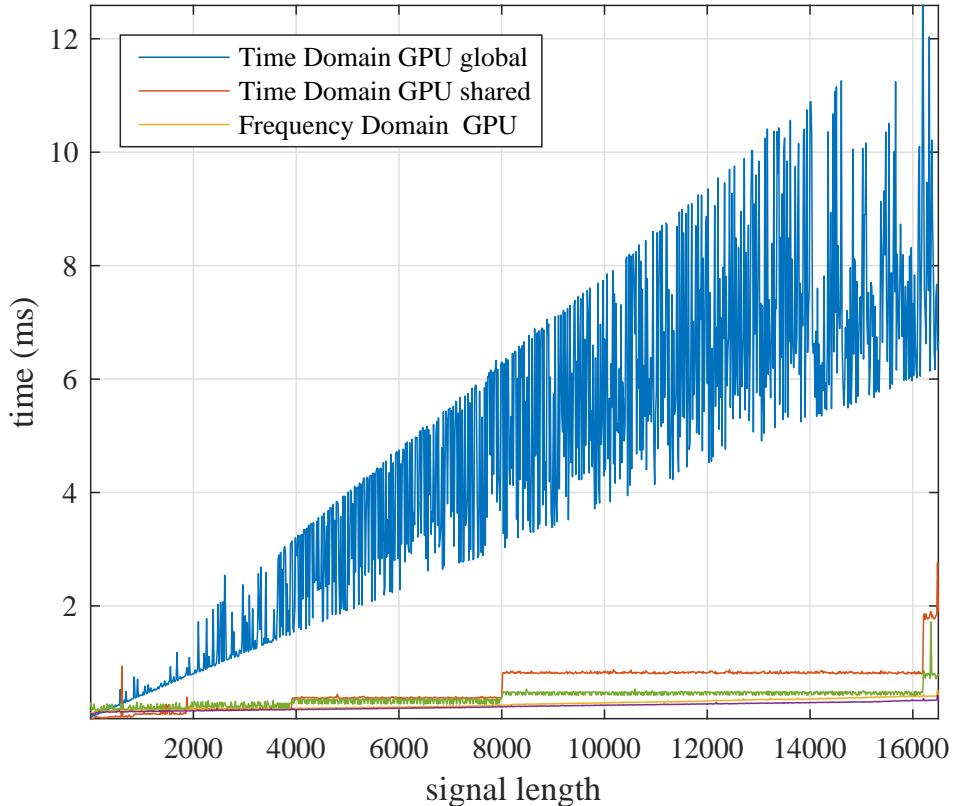
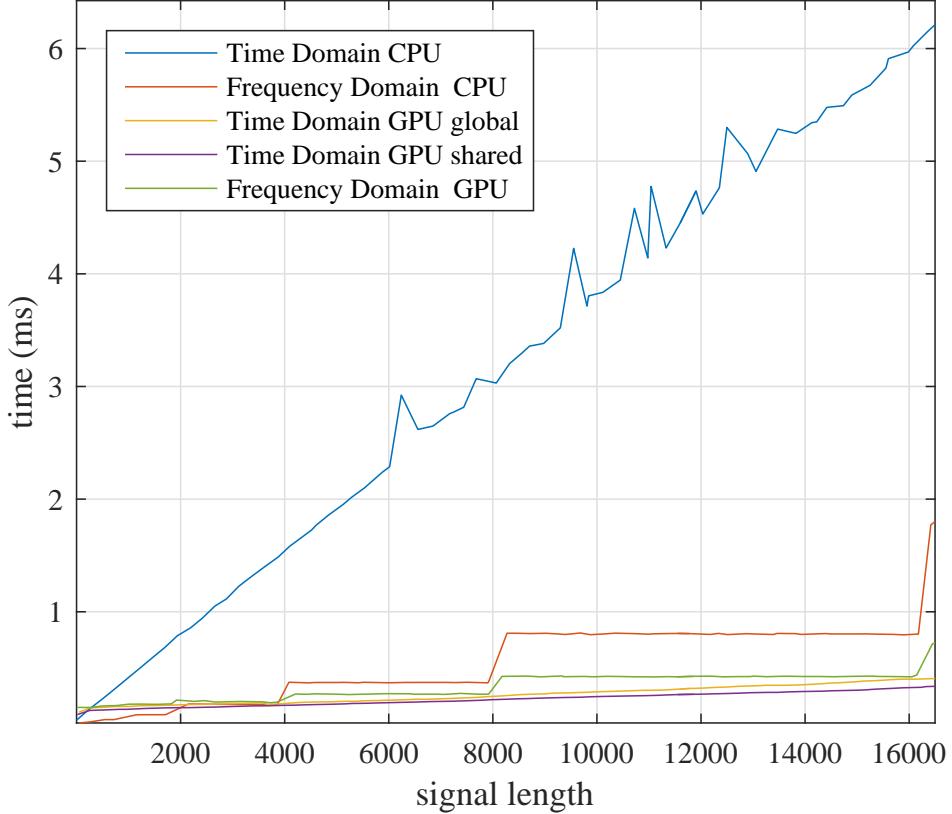


Figure 5.6: Comparison of a complex convolution on CPU verse GPU. The signal length is varied and the filter is fixed at 186 taps. A lower bound was applied by searching for a local minimums in 15 sample width windows.

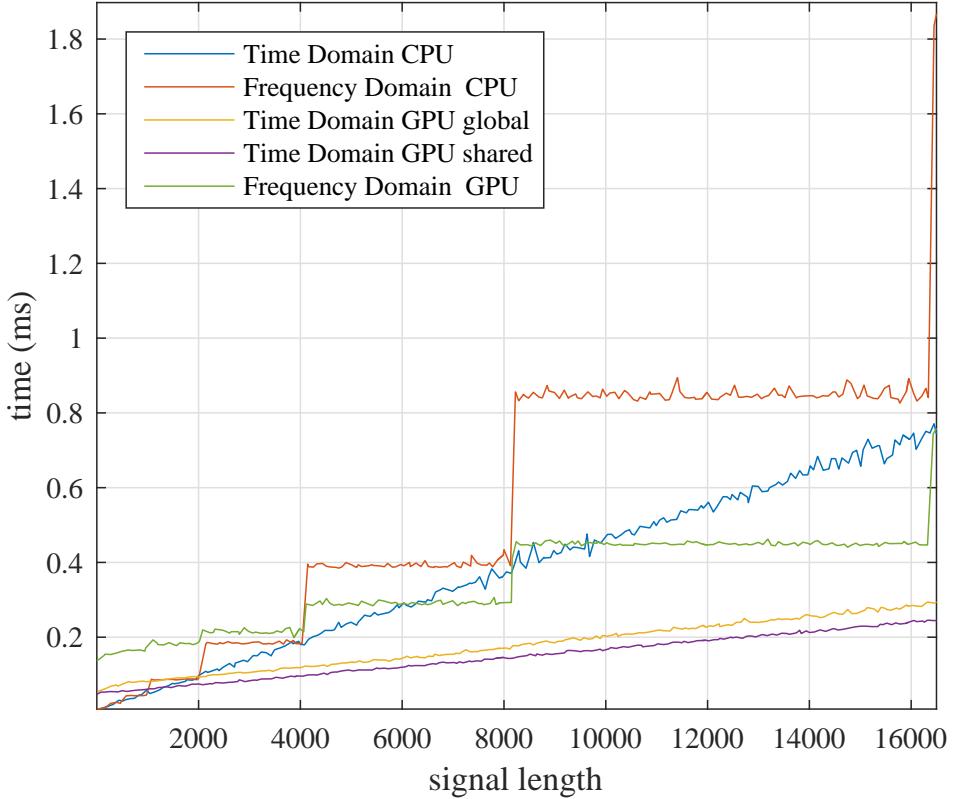


of flops predicted. The GPU does have the stair stepping from appending zeros for the frequency domain, but the time domain GPU kernels execute in less time.

The GPU execution time does not follow the same trend as the number of flops. Why? As mentioned in Section 4.3, GPUs have a hanes amount of computational resources and limited memory bandwidth. Over 90% of GPU kernels are memory bandwidth limited.

To provide more proof, compare Figures 5.7 and 5.4. 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. On “short” filters, using shared memory will perform better than using only global memory. Using shared memory to store the short filter saves the each iteration a lot of execution time.

Figure 5.7: 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.

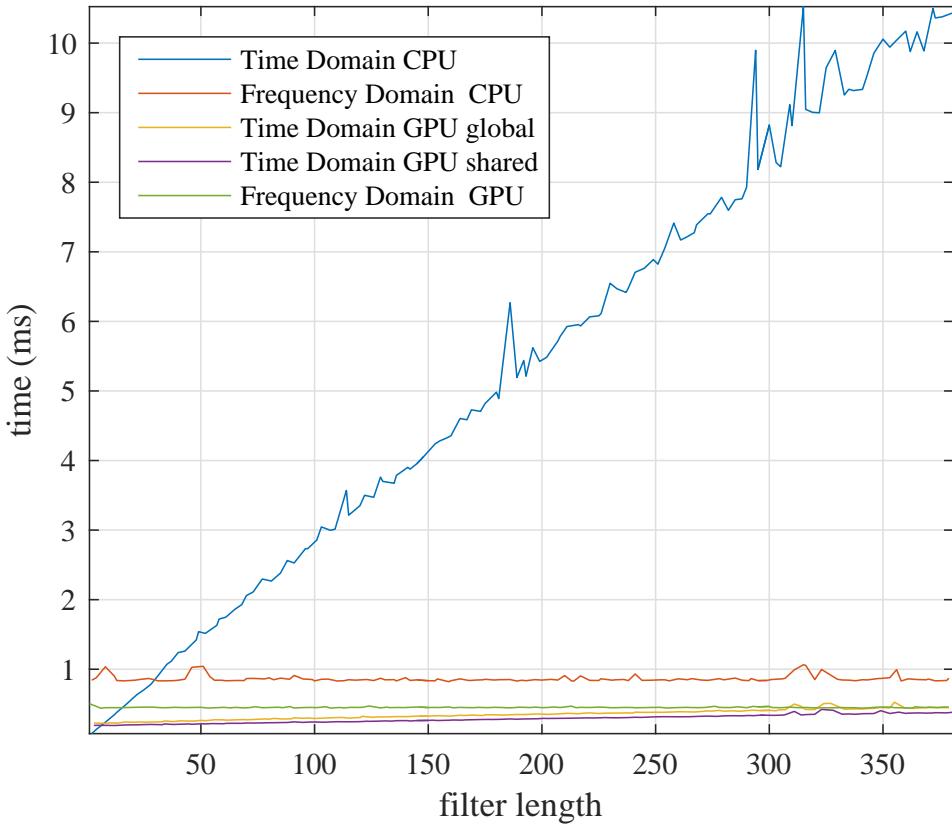


What if the signal length was set and the filter length was varied? Figure 5.8 compares CPU to GPU execution time of a 12672 sample signal convolved with a varying filter length. The time domain CPU execution time is affected obviously because the number of flops increases.

Neither CPU or GPU frequency domain execution time is affected by varying filter length. The execution time stays the same because the number of memory accesses and flops remain constant because the filter is appended with zeros to the convolution 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 5.4 and 5.5 suggest.

Figure 5.8: 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.



Conclusion, when needing implement convolution in a CPU or GPU, implement it every way possible. Which ever way has the fastest execution time, he is your winner. Long story short, most of the time a GPU DSP engineer is given a set length for the signal and filter. As Figures 5.5 through 5.8 have shown, unless every implementation is explored, there is no way of saying which implementation will absolutely be fastest.

Table 5.2 shows GPU frequency domain is fastest when convolving a 12672 sample signal with a 186 tap filter. Table 5.3 shows GPU time domain using shared is fastest when convolving a 12672 sample signal with a 21 tap filter.

Table 5.2: 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	5.0388
CPU frequency domain	FFTW	1.6295
GPU time domain global	ConvGPU	0.4021
GPU time domain shared	ConvGPUshared	0.3752
GPU frequency domain	cuFFT	0.3387

Table 5.3: 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	0.6125
CPU frequency domain	FFTW	2.5947
GPU time domain global	ConvGPU	0.2405
GPU time domain shared	ConvGPUshared	0.2112
GPU frequency domain	cuFFT	0.3360

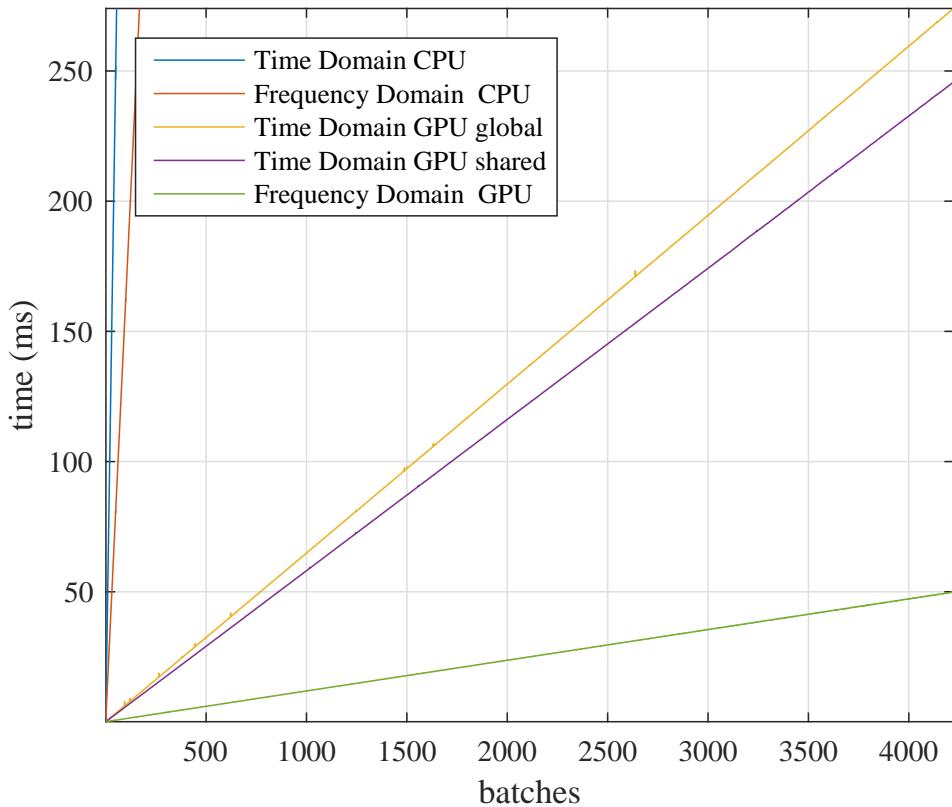
5.4 Batched Convolution

In section 5.3 convolution of a single signal with a single filter was studied. Chapter blah (system overview) shows the packetized structure of the received signal. The received signal has 3104 packets or batches and each packet is independent of other packets.

Now that we have 3104 signals to be convolved with 3104 filters, how does the problem change? Which approach to convolution will be fastest?

As the number of batches increases, does CPU and GPU execution time increase linearly? Figure 5.9 shows how the execution time increases with the number of batches. 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

Figure 5.9: 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.



CPU time and frequency domain execution time skyrockets as the number of batches increases.

The GPU handles batched processing very well because it introduces more parallelism.

Judging by Figure 5.9, CPU is not a contender in batched processing for fast execution times when compared to the GPU. CPU and GPU batched processing will not be compared any further. Listing 5.2 shows three ways 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 5.4 shows how Listing 5.2 is timed.

Table 5.4: Defining start and stop lines for timing comparison in Listing 5.2.

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

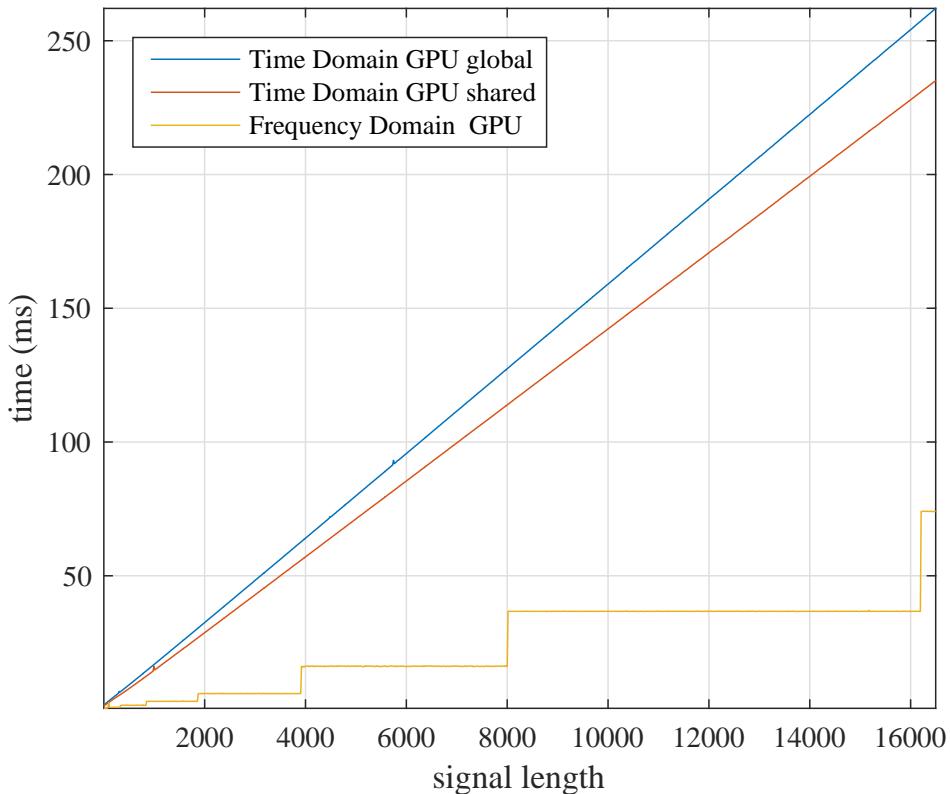


Figure 5.10 shows execution time for 3104 batches of 186 tap filters convolved with varying signal lengths. Performing frequency domain convolution is always faster than time domain convolution because the cufft library is better optimized for batched processing. Frequency domain convolution for a 12672 sample signal takes just 36.8ms, that is on average 0.0119ms per batch. Compare 0.0119ms per batch to single batched execution time in Table 5.2, one batch took 0.3387. Batched processing introduced a 28× speed up!

Figure 5.11: Comparison of a batched complex convolution on a GPU. The signal length is varied and the filter is fixed at 21 taps.

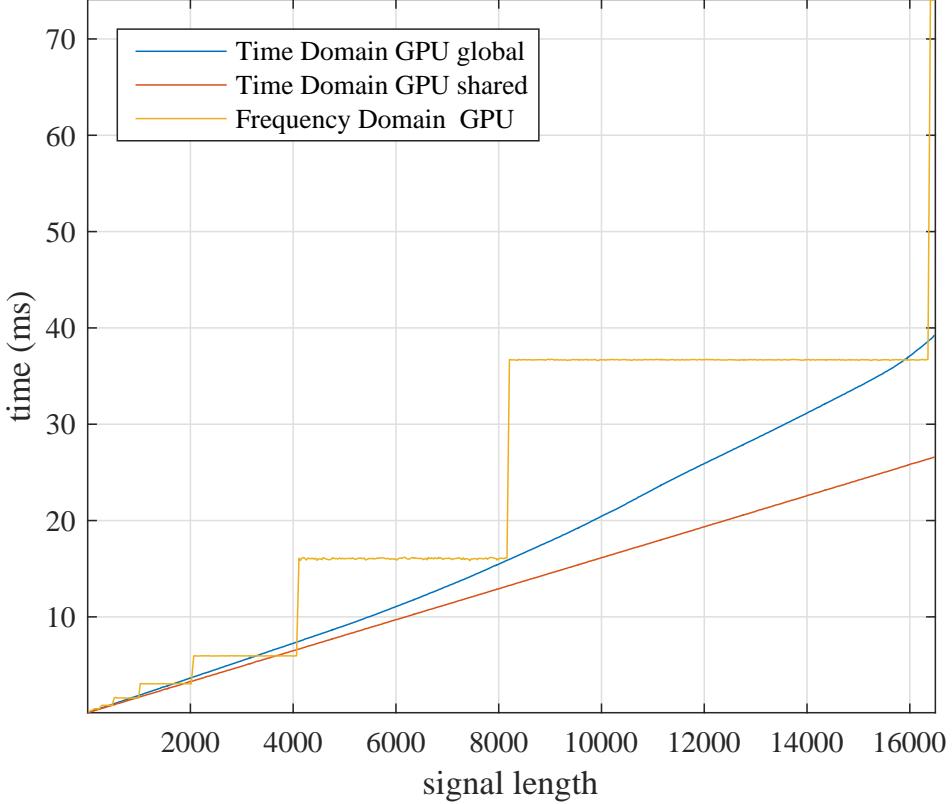
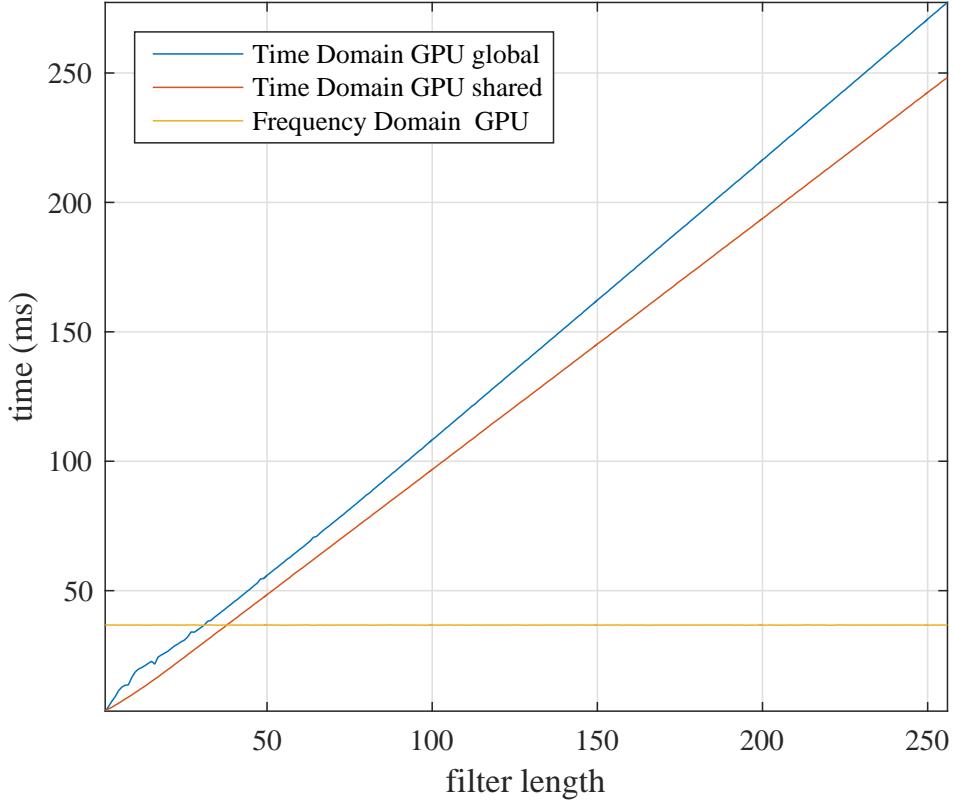


Figure 5.11 shows execution time for 3104 batches of 21 tap filters convolved with varying signal lengths. This figure exhibits the same characteristics of single batch convolution execution time shown in Figure 5.7. For most signal lengths, performing time domain convolution using shared memory is fastest.

Figure 5.12 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 5.8 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 that in batched processing the algorithm leading to the fastest execution time still depends on signal and filter length, one important concept has been overlooked. Figure 3.2 shows there are two filters that need to be allied to the signal.

Figure 5.12: 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 ConvGPU or ConvGPUshared performs the convolution of the 186 tap equalizer and 21 detection filter. The second call of ConvGPU or ConvGPUshared performs the 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 must be changed from two input vectors 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 in Figure 3.2 constant. The FFT of the detection filter can be calculated and stored at initialization.

Table 5.5 shows the batched convolution execution time for a 12672 sample signal and 186 tap filter. Table 5.6 shows the batched convolution execution time for a 12672 sample signal and

Table 5.5: 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	ConvGPUsShared	180.272
GPU frequency domain	cuFFT	36.798

Table 5.6: 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	ConvGPUsShared	20.4287
GPU frequency domain	cuFFT	36.7604

21 tap filter. Table 5.8 shows the batched cascaded convolution execution time for a 12672 sample signal with 21 and 186 tap filters.

Tables 5.5 and 5.6 agree with Figures 5.11 and 5.10. Time domain convolution is faster with a short 21 tap filter but frequency domain convolution is faster with a long 186 tap filter.

Figure 5.13 shows two ways to cascade the signal r through two filters. Rather than applying both filters to the signal, compute a shorter convolution of the 186 and 21 tap filters then apply the $186 + 21 - 1$ tap result to the signal.

Table 5.7: 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	ConvGPUsShared	199.844
GPU frequency domain	cuFFT	36.7704

Table 5.8: Batched convolution execution times with for a 12672 sample signal and cascaded 21 and 186 tap filter on a Tesla K40c GPU.

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

Figure 5.13: Two ways to convolve the signal r with the 186 tap filter c and 21 tap filter d .

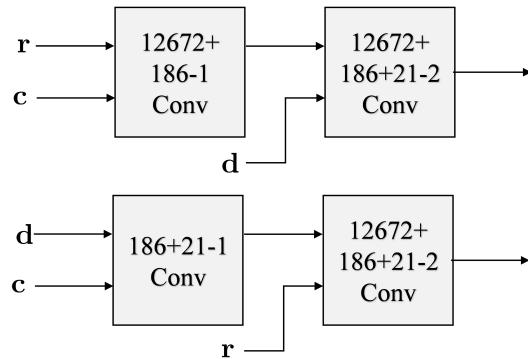


Table 5.8 shows the execution time of implementing cascaded filters, convolving the 21 and 186 tap filters is extremely fast in the GPU. It only costs 2.3165ms to apply an extra filter in the frequency domain. It costs 22.0170ms and 19.7460ms to apply an extra filter in the time domain because the cascaded filter is now 206 taps rather than 186. Table 5.7 confirms it costs an extra 20ms or so to apply a 206 vs 186 tap filter.

Listing 5.1: 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.

```

1 #include <iostream>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <cufft.h>
5 #include <fstream>
6 #include <string>
7 #include <fftw3.h>
8 using namespace std;
9
10
11 void ConvCPU(cufftComplex* y, cufftComplex* x, cufftComplex* h, int Lx, int Lh) {
12     for(int yIdx = 0; yIdx < Lx+Lh-1; yIdx++) {
13         cufftComplex temp;
14         temp.x = 0;
15         temp.y = 0;
16         for(int hIdx = 0; hIdx < Lh; hIdx++) {
17             int xAccessIdx = yIdx-hIdx;
18             if(xAccessIdx>=0 && xAccessIdx<Lx) {
19                 // temp += x[xAccessIdx]*h[hIdx];
20                 float A = x[xAccessIdx].x;
21                 float B = x[xAccessIdx].y;
22                 float C = h[hIdx].x;
23                 float D = h[hIdx].y;
24                 cufftComplex result;
25                 result.x = A*C-B*D;
26                 result.y = A*D+B*C;
27                 temp.x += result.x;
28                 temp.y += result.y;
29             }
30         }
31         y[yIdx] = temp;
32     }
33 }
34
35
36 __global__ void ConvGPU(cufftComplex* y, cufftComplex* x, cufftComplex* h, int Lx, int Lh) {
37     int yIdx = blockIdx.x*blockDim.x + threadIdx.x;
38
39     int lastThread = Lx+Lh-1;
40
41     // don't access elements out of bounds
42     if(yIdx >= lastThread)
43         return;
44
45     cufftComplex temp;
46     temp.x = 0;
47     temp.y = 0;
48     for(int hIdx = 0; hIdx < Lh; hIdx++) {
49         int xAccessIdx = yIdx-hIdx;
50         if(xAccessIdx>=0 && xAccessIdx<Lx) {
51             // temp += x[xAccessIdx]*h[hIdx];
52             float A = x[xAccessIdx].x;
53             float B = x[xAccessIdx].y;
54             float C = h[hIdx].x;
55             float D = h[hIdx].y;
56             cufftComplex result;
57             result.x = A*C-B*D;
58             result.y = A*D+B*C;
59             temp.x += result.x;
60             temp.y += result.y;
61         }
62     }
63     y[yIdx] = temp;
64 }
```

```

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

```

```

133
134 int main(){
135     int 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 /**

```

```

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

```

```
325     cudaMemcpy(myConv5, dev_mySignal5, Nfft*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
326
327     cufftDestroy(plan);
328
329     free(mySignal1);
330     free(mySignal2);
331
332     free(myFilter1);
333     free(myFilter2);
334
335     free(myConv1);
336     free(myConv2);
337     free(myConv2_timeReversed);
338     free(myConv3);
339     free(myConv4);
340     free(myConv5);
341
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 5.2: 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.

```

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

```

```

64     temp.y = 0;
65     for(int hIdx = 0; hIdx < Lh; hIdx++){
66         int xAccessIdx = yIdx-hIdx;
67         if(xAccessIdx>=0 && xAccessIdx<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 numThreadsPerBlock;
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             ));
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             ));
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,
196               cudaMemcpyHostToDevice);

```

```

196     cudaMemcpy(dev_myFilter1, myFilter1, numBatches*sizeof(cufftComplex)*myFilterLength,
197                 cudaMemcpyHostToDevice);
198
199     maxThreads = myConvLength*numBatches;
200     numTreadsPerBlock = 128;
201     numBlocks = maxThreads/numTreadsPerBlock;
202     if(maxThreads % numTreadsPerBlock > 0)
203         numBlocks++;
204     ConvGPU<<<numBlocks, numTreadsPerBlock>>>(dev_myConv1, dev_mySignal1, dev_myFilter1,
205             mySignalLength, myFilterLength, maxThreads);
206
207     cudaMemcpy(myConv1, dev_myConv1, myConvLength*numBatches*sizeof(cufftComplex),
208                 cudaMemcpyDeviceToHost);
209
210     /**
211      * Time Domain Convolution GPU Using Shared Memory
212      */
213     cudaMemcpy(dev_mySignal2, mySignal1, numBatches*sizeof(cufftComplex)*mySignalLength,
214                 cudaMemcpyHostToDevice);
215     cudaMemcpy(dev_myFilter2, myFilter1, numBatches*sizeof(cufftComplex)*myFilterLength,
216                 cudaMemcpyHostToDevice);
217
218     maxThreads = myConvLength*numBatches;
219     numTreadsPerBlock = 256;
220     numBlocks = maxThreads/numTreadsPerBlock;
221     if(maxThreads % numTreadsPerBlock > 0)
222         numBlocks++;
223     ConvGPUSHared<<<numBlocks, numTreadsPerBlock, myFilterLength*sizeof(cufftComplex)>>>(
224         dev_myConv2, dev_mySignal2, dev_myFilter2, mySignalLength, myFilterLength,maxThreads)
225         ;
226
227     cudaMemcpy(myConv2, dev_myConv2, myConvLength*numBatches*sizeof(cufftComplex),
228                 cudaMemcpyDeviceToHost);
229
230     /**
231      * Frequency Domain Convolution GPU
232      */
233     cudaMemcpy(dev_mySignal3, mySignal1_pad, Nfft*numBatches*sizeof(cufftComplex),
234                 cudaMemcpyHostToDevice);
235     cudaMemcpy(dev_myFilter3, myFilter1_pad, Nfft*numBatches*sizeof(cufftComplex),
236                 cudaMemcpyHostToDevice);
237
238     cufftExecC2C(plan, dev_mySignal3, dev_mySignal3, CUFFT_FORWARD);
239     cufftExecC2C(plan, dev_myFilter3, dev_myFilter3, CUFFT_FORWARD);
240
241     maxThreads = Nfft*numBatches;
242     numTreadsPerBlock = 96;
243     numBlocks = maxThreads/numTreadsPerBlock;
244     if(maxThreads % numTreadsPerBlock > 0)
245         numBlocks++;
246     PointToPointMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal3, dev_myFilter3,
247         maxThreads);
248     cufftExecC2C(plan, dev_mySignal3, dev_mySignal3, CUFFT_INVERSE);
249
250     numTreadsPerBlock = 640;
251     numBlocks = maxThreads/numTreadsPerBlock;
252     if(maxThreads % numTreadsPerBlock > 0)
253         numBlocks++;
254     float scalar = 1.0/((float)Nfft);
255     ScalarMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal3, scalar, maxThreads);
256
257     cudaMemcpy(myConv3, dev_mySignal3, Nfft*numBatches*sizeof(cufftComplex),
258                 cudaMemcpyDeviceToHost);
259
260     cufftDestroy(plan);
261
262     // Free vectors on CPU
263     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 6

Equalizer Equations

6.1 Overview

This thesis examines the 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 very similar also with one subtle difference. CMA isn't related to any other algorithm aside from being initialized to MMSE.

6.2 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

$$\mathbf{R}\mathbf{c} = \hat{\mathbf{h}} \tag{6.1}$$

where \mathbf{c} is vector of the desired equalizer coefficients and \mathbf{R} is the 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} .

6.2.1 Zero-Forcing

The ZF equalizer is an FIR filter defined by the coefficients

$$c_{\text{ZF}}(-L_1) \quad \cdots \quad c_{\text{ZF}}(0) \quad \cdots \quad c_{\text{ZF}}(L_2). \quad (6.2)$$

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

$$\mathbf{c}_{\text{ZF}} = (\mathbf{H}^\dagger \mathbf{H})^{-1} \mathbf{H}^\dagger \mathbf{u}_{n_0} \quad (6.3)$$

where

$$\mathbf{c}_{\text{ZF}} = \begin{bmatrix} c_{\text{ZF}}(-L_1) \\ \vdots \\ c_{\text{ZF}}(0) \\ \vdots \\ c_{\text{ZF}}(L_2) \end{bmatrix}, \quad (6.4)$$

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \left. \right\} \begin{array}{l} n_0 - 1 \text{ zeros} \\ \\ \\ \\ N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \end{array}, \quad (6.5)$$

and

$$\mathbf{H} = \begin{bmatrix} \hat{h}(-N_1) & & & \\ \hat{h}(-N_1 + 1) & \hat{h}(-N_1) & & \\ \vdots & \vdots & \ddots & \\ \hat{h}(N_2) & \hat{h}(N_2 - 1) & \hat{h}(-N_1) & \\ & \hat{h}(N_2) & \hat{h}(-N_1 + 1) & \\ & & \vdots & \\ & & & \hat{h}(N_2) \end{bmatrix}. \quad (6.6)$$

Calculating Equation (6.3) would give the desired result but the computation is heavy. 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 (6.3) results in

$$\begin{aligned} \mathbf{H}^\dagger \mathbf{H} \mathbf{c}_{\text{ZF}} &= \mathbf{H}^\dagger \mathbf{u}_{n_0} \\ \mathbf{R}_{\hat{h}} \mathbf{c}_{\text{ZF}} &= \hat{\mathbf{h}}_{n_0} \end{aligned} \quad (6.7)$$

where

$$\mathbf{R}_{\hat{h}} = \mathbf{H}^\dagger \mathbf{H} = \begin{bmatrix} r_{\hat{h}}(0) & r_{\hat{h}}^*(1) & \cdots & r_{\hat{h}}^*(L_{eq} - 1) \\ r_{\hat{h}}(1) & r_{\hat{h}}(0) & \cdots & r_{\hat{h}}^*(L_{eq} - 2) \\ \vdots & \vdots & \ddots & \\ r_{\hat{h}}(L_{eq} - 1) & r_{\hat{h}}(L_{eq} - 2) & \cdots & r_{\hat{h}}(0) \end{bmatrix} \quad (6.8)$$

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

$$r_{\hat{h}}(k) = \sum_{n=-N_1}^{N_2} \hat{h}(n) \hat{h}^*(n - k). \quad (6.9)$$

and

$$\hat{\mathbf{h}}_{n_0} = \mathbf{H}^\dagger \mathbf{u}_{n_0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix} \quad (6.10)$$

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

Note that $\mathbf{R}_{\hat{h}}$ can be built by computing

$$\mathbf{r}_{\hat{h}} = \begin{bmatrix} r_{\hat{h}}(0) \\ \vdots \\ r_{\hat{h}}(L_{ch}) \\ r_{\hat{h}}(L_{ch} + 1) \\ \vdots \\ r_{\hat{h}}(L_{eq} - 1) \end{bmatrix} = \begin{bmatrix} r_{\hat{h}}(0) \\ \vdots \\ r_{\hat{h}}(L_{ch}) \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad (6.11)$$

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

MMSE Equalizer

The MMSE equalizer is an FIR filter defined by the coefficients

$$c_{\text{MMSE}}(-L_1) \quad \cdots \quad c_{\text{MMSE}}(0) \quad \cdots \quad c_{\text{MMSE}}(L_2). \quad (6.12)$$

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

$$\mathbf{c}_{MMSE} = [\mathbf{G}\mathbf{G}^\dagger + 2\hat{\sigma}_w^2 \mathbf{I}_{L_1+L_2+1}]^{-1} \mathbf{g}^\dagger \quad (6.13)$$

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

$$\mathbf{G} = \begin{bmatrix} \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \\ & \ddots & \\ \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \\ & \ddots & \ddots \\ & & \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \end{bmatrix} \quad (6.14)$$

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

$$\mathbf{g}^\dagger = \hat{\mathbf{h}}_{n0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix}. \quad (6.15)$$

Computing \mathbf{c}_{MMSE} can be simplified by noticing that $\mathbf{g}^\dagger = \hat{\mathbf{h}}_{n0}$, $\mathbf{G}\mathbf{G}^\dagger = \mathbf{R}_{\hat{h}}$ in Equation (6.8) and defining

$$\mathbf{R}_{\hat{h}w} = \mathbf{R}_{\hat{h}} + 2\hat{\sigma}_w^2 \mathbf{I}_{L_1+L_2+1} = \begin{bmatrix} r_h(0) + 2\hat{\sigma}_w^2 & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) + 2\hat{\sigma}_w^2 & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) + 2\hat{\sigma}_w^2 \end{bmatrix}. \quad (6.16)$$

By placing Equation (6.16) and (6.15) into (6.13), solving for the MMSE equalizer coefficients \mathbf{c}_{MMSE} takes the form like the ZF equalizer coeffiencts in (6.7)

$$\mathbf{R}_{\hat{h}w} \mathbf{c}_{\text{MMSE}} = \hat{\mathbf{h}}_{n0}. \quad (6.17)$$

The only difference between solving for the ZF or MMSE equalizer coefficients is $\mathbf{R}_{\hat{h}w}$ or $\mathbf{R}_{\hat{h}}$. MMSE equalizer uses the noise variance estimate by adding $2\hat{\sigma}_w^2$ to $r_h(0)$ 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 $\mathbf{R}_{\hat{h}}$.

6.2.2 The Constant Modulus Algorithm

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

$$c_{\text{CMA}(b)}(-L_1) \quad \dots \quad c_{\text{CMA}(b)}(0) \quad \dots \quad c_{\text{CMA}(b)}(L_2). \quad (6.18)$$

The filter coefficients are calculated by a steepest decent algorithm

$$\mathbf{c}_{\text{CMA}(b+1)} = \mathbf{c}_{\text{CMA}(b)} - \mu \nabla J \quad (6.19)$$

initialized by the MMSE equalizer coefficients

$$\mathbf{c}_{\text{CMA}(0)} = \mathbf{c}_{\text{MMSE}}. \quad (6.20)$$

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

$$\nabla J \approx \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \left[y(n)y^*(n) - 1 \right] y(n)\mathbf{r}^*(n). \quad (6.21)$$

where

$$\mathbf{r}(n) = \begin{bmatrix} r(n + L_1) \\ \vdots \\ r(n) \\ \vdots \\ r(n - L_2) \end{bmatrix}. \quad (6.22)$$

This means ∇J is defined by

$$\nabla J = \begin{bmatrix} \nabla J(-L_1) \\ \vdots \\ \nabla J(0) \\ \vdots \\ \nabla J(L_2) \end{bmatrix}. \quad (6.23)$$

A DSP engineer could implement Equation (6.21) and (6.19) directly but CMA can be massaged to map better to GPUs. To leverage the computational efficiency of convolution using the cuFFT library, Equation (6.21) is re-expressed as a convolution.

To begin messaging ∇J

$$z(n) = 2 \left[y(n)y^*(n) - 1 \right] y(n) \quad (6.24)$$

is defined to make the expression of ∇J to be

$$\nabla J = \frac{1}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} z(n) \mathbf{r}^*(n). \quad (6.25)$$

then writing the summation out in vector form

$$\nabla J = \frac{z(0)}{L_{pkt}} \begin{bmatrix} r^*(L_1) \\ \vdots \\ r^*(0) \\ \vdots \\ r^*(L_2) \end{bmatrix} + \frac{z(1)}{L_{pkt}} \begin{bmatrix} r^*(1+L_1) \\ \vdots \\ r^*(1) \\ \vdots \\ r^*(1-L_2) \end{bmatrix} + \dots + \frac{z(L_{pkt}-1)}{L_{pkt}} \begin{bmatrix} r^*(L_{pkt}-1+L_1) \\ \vdots \\ r^*(L_{pkt}-1) \\ \vdots \\ r^*(L_{pkt}-1-L_2) \end{bmatrix}. \quad (6.26)$$

The k th value of ∇J is

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m) r^*(m-k), \quad -L_1 \leq k \leq L_2. \quad (6.27)$$

The summation almost looks like a convolution. To put the summation in convolution form, define

$$\rho(n) = r^*(n). \quad (6.28)$$

Now

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)\rho(k-m). \quad (6.29)$$

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

$$\begin{aligned} b(n) &= \sum_{m=0}^{L_{pkt}-1} z(m)\rho(n-m) \\ &= \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-n) \end{aligned} \quad (6.30)$$

Comparing Equation (6.29) and (6.30) shows that

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2. \quad (6.31)$$

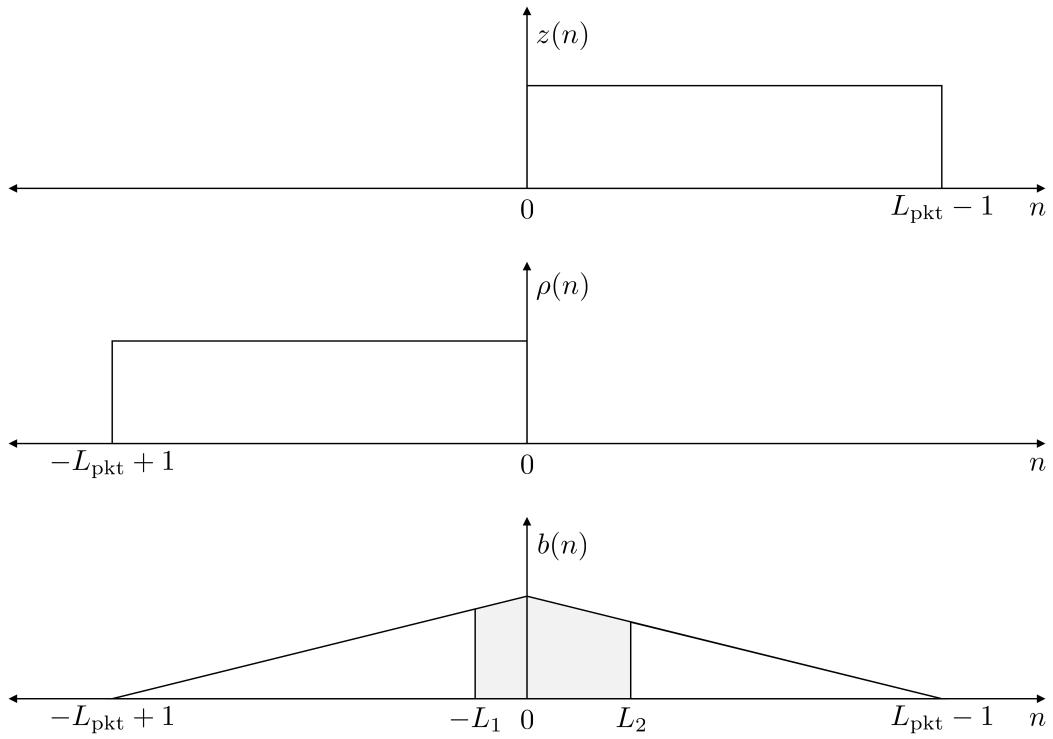
The values of interest are shown in Figure 6.1.

This suggest the following algorithm for computing the gradient vector ∇J .

Listing 6.1: Matlab code showing how to compute implement CMA through convolution using FFTs.

```
c_CMA = c_MMSE;
for i = 1:its
    yy = conv(r,c_CMAb);
    y = yy(L1+1:end-L2); % trim yy
    z = 2*(y.*conj(y)-1).*y;
    Z = fft(z,Nfft);
    R = fft(conj(r(end:-1:1)),Nfft)
    b = ifft(Z.*R);
    delJ = b(Lpkt-L1:Lpkt+L2);
    c_CMAb1 = c_CMAb -mu*delJ;
    c_CMAb = c_CMAb1;
```

Figure 6.1: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.



```

end
yy = conv(r,c_CMA);
y = yy(L1+1:end-L2); % trim yy

```

6.2.3 The Frequency Domain Equalizers

Williams and Saquib derived the Frequency Domain Equalizers (FDEs) for this application [11,].

Frequency Domain Equalizer One (FDE1) is the MMSE or Wiener filter applied in the frequency domain. FDE1 is adapted from Williams' and Saquib's Frequency Domain Equalizers [11, eq. (11)]

$$C_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1. \quad (6.32)$$

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

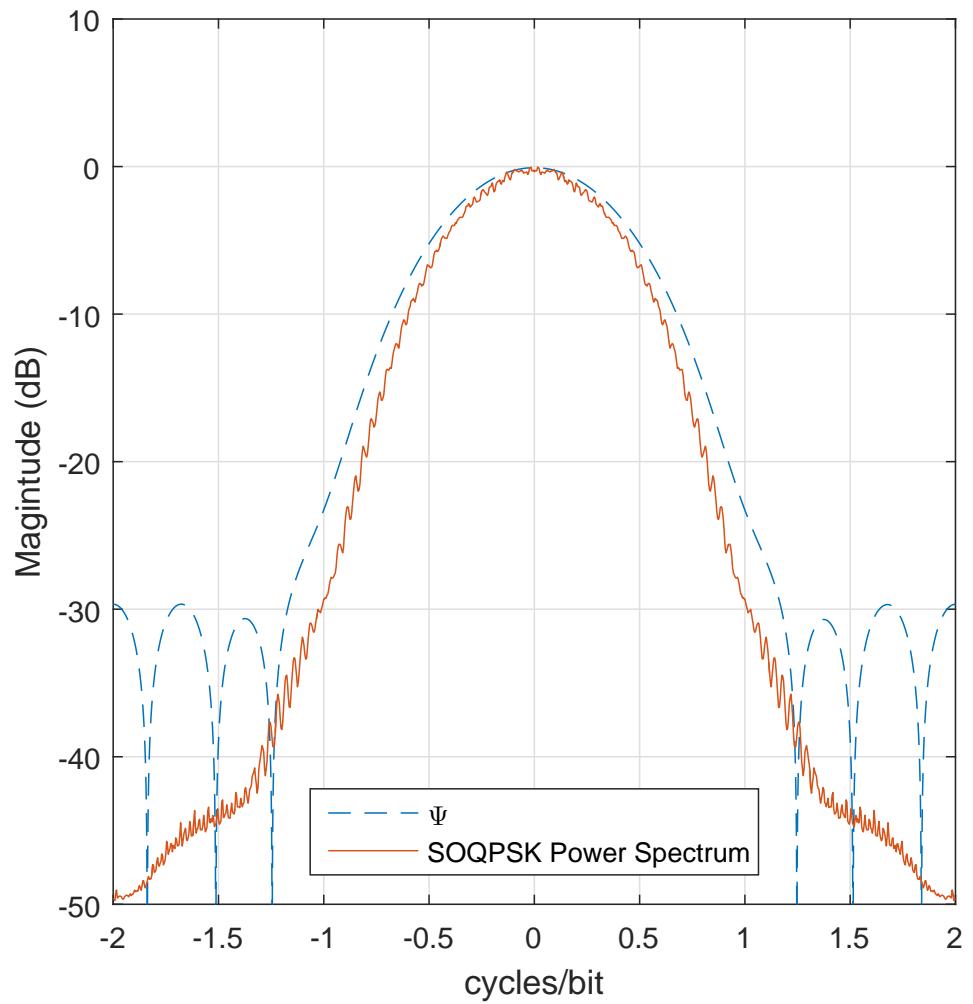
Equation (6.32) is straight forward to implement in GPUs. FDE1 is extremely fast and computationally efficient.

Frequency Domain Equalizer Two (FDE2) is also the MMSE or Wiener filter applied in the frequency domain but knowledge of the SOQPSK-TG spectrum is leveraged. FDE1 is adapted from Williams' and Saquib's Frequency Domain Equalizers [11, eq. (12)] The FDE2 equalizer is defined in Equation (12) as

$$C_{\text{FDE2}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1 \quad (6.33)$$

FDE2 almost identical to FDE1. The only difference is the term $\Psi(e^{j\omega_k})$ in the denominator. The term $\Psi(\omega)$ is the averaged SPQOSK-TG power spectrum shown in Figure 6.2.

Figure 6.2: I need help on this one!!!!



Chapter 7

Equalizer GPU Implementation

Each equalizer presents an interesting challenge from a GPU implementation perspective. This is why in Chapter 6 the equalizer equations were massaged and conditioned. This chapter will explain how the equalizers were computed and applied in batch processing.

Chapter 5.2 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 coefficient vectors.

Convolution is used many times in this chapter. Frequency domain convolution has many little steps that make a block diagram look busy. Figures 7.1 and 7.2 show how frequency domain convolution will be represented as one block from in this chapter.

7.1 Zero-Forcing and MMSE GPU Implementation

The ZF and MMSE equalizer coefficient computations have exactly the same form as shown in Equations 6.7 and 6.17

$$\mathbf{R}_{\hat{h}} \mathbf{c}_{\text{ZF}} = \hat{\mathbf{h}}_{n_0} \quad (7.1)$$

$$\mathbf{R}_{\hat{h}w} \mathbf{c}_{\text{MMSE}} = \hat{\mathbf{h}}_{n_0}. \quad (7.2)$$

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 7.1: Convolution of vectors \mathbf{c} and \mathbf{r} block diagram simplified to one block marked Conv.

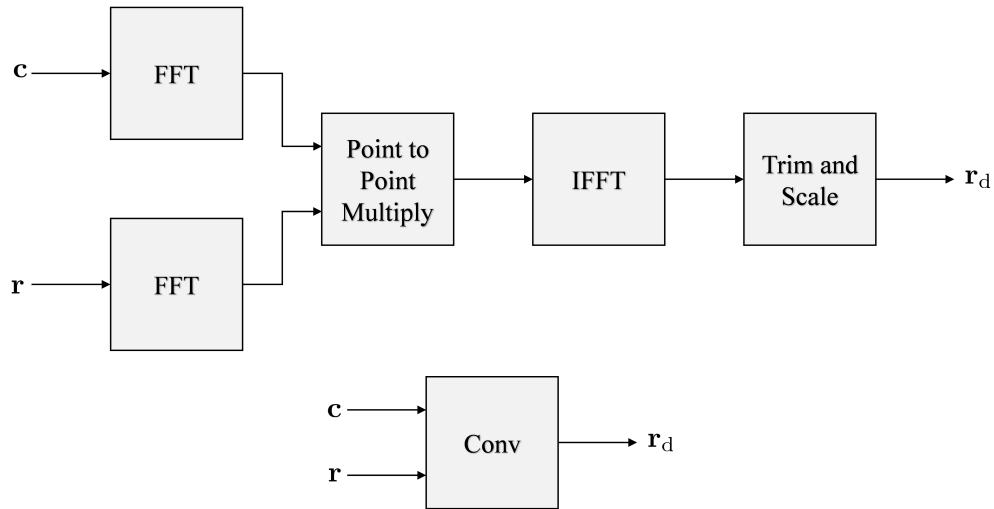
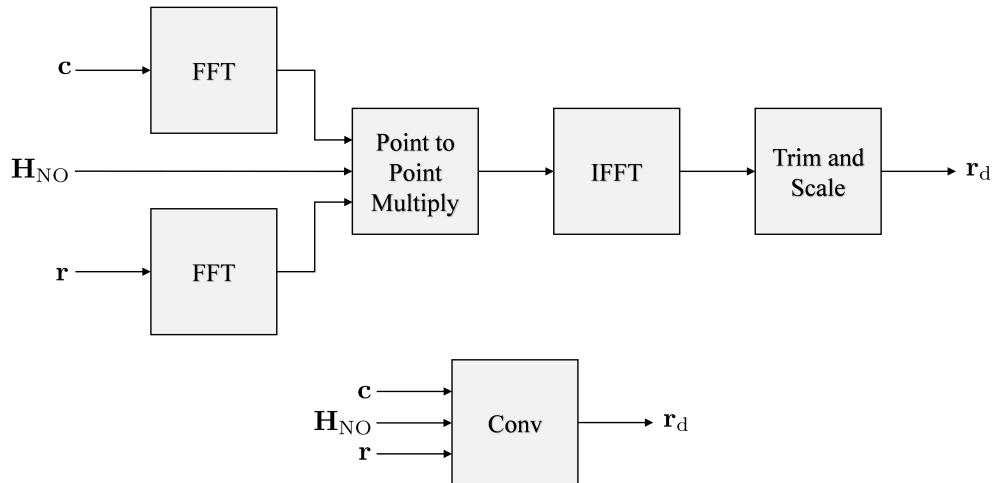


Figure 7.2: Convolution of vectors \mathbf{c} , \mathbf{r} and \mathbf{H}_{NO} block diagram simplified to one block marked Conv.



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

$$\mathbf{c}_{ZF} = \mathbf{R}_{\hat{h}}^{-1} \hat{\mathbf{h}}_{n_0} \quad (7.3)$$

$$\mathbf{c}_{MMSE} = \mathbf{R}_{\hat{h}w}^{-1} \hat{\mathbf{h}}_{n_0}. \quad (7.4)$$

Before solving Equation 7.3, $\mathbf{R}_{\hat{h}}$ 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 6.9 directly in one GPU kernel. The computation of $\mathbf{r}_{\hat{h}}$ is very fast because the length of the channel estimate L_{ch} is very short.

Section 6.2.1 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}_{ZF} in less than 500ms.

The sparseness of $\mathbf{R}_{\hat{h}}$ is leveraged using a sparse solver function called “cusolverSpCcsqrsvBatched”. cusolverSpCcsqrsvBatched 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 186×186 matrix $\mathbf{R}_{\hat{h}}$ to a 12544 element CSR matrix $\mathbf{R}_{\hat{h}_{\text{CRS}}}$. Before cusolverSpCcsqrsvBatched can be called, the CSR matrix $\mathbf{R}_{\hat{h}_{\text{CRS}}}$ has to be built using $\mathbf{r}_{\hat{h}}$. An example of how to use the CUDA cusolverSp library can be found [6].

Figure 7.3 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 $\mathbf{R}_{\hat{h}_{\text{CRS}}}$, $\hat{\sigma}_w^2$ is added to the main diagonal elements.

Using cusolverSpCcsqrsvBatched wasn't the only implementation researched. Table 7.1 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 $\mathbf{R}_{\hat{h}_{\text{CRS}}}$ [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}_{ZF} or \mathbf{c}_{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 7.3: Block Diagram showing how the Zero-Forcing equalizer coefficients are implemented in the GPU.

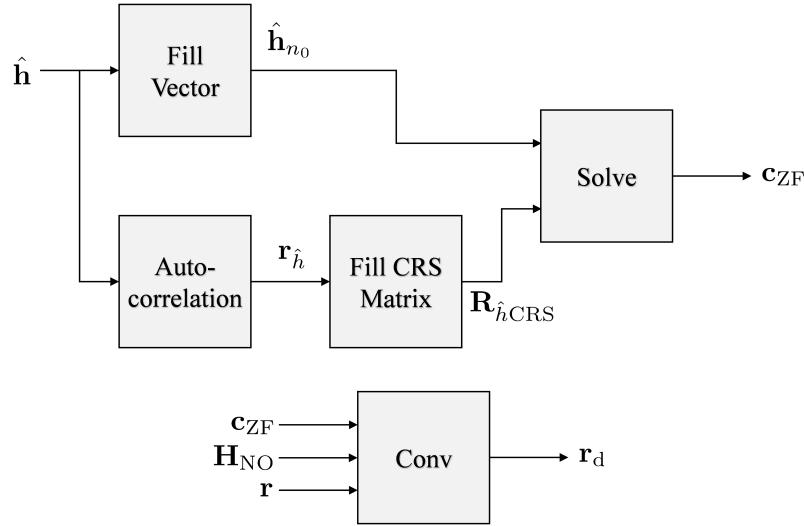


Figure 7.4: Block Diagram showing how the Minimum Mean Squared Error equalizer coefficients are implemented in the GPU.

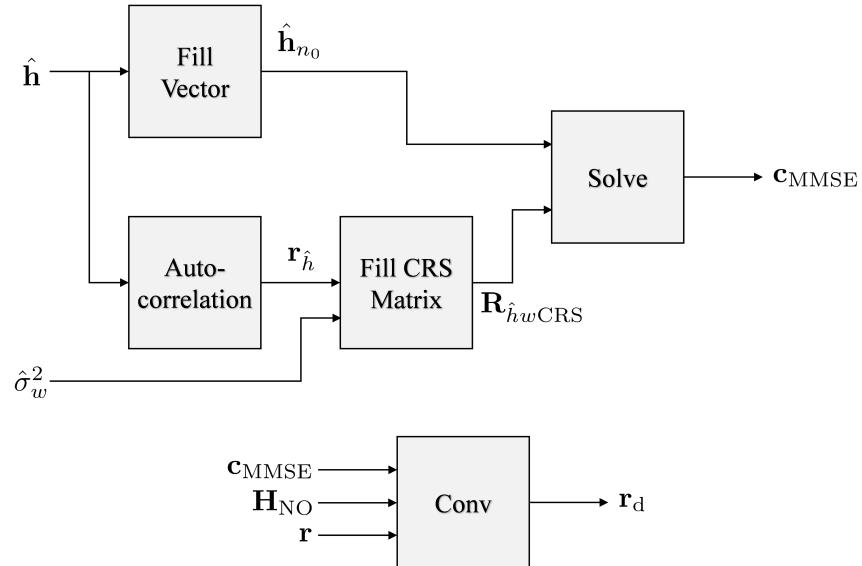


Table 7.1: Defining start and stop lines for timing comparison in Listing 5.1.

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

7.2 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 because CMA is a steepest decent.

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

$$\mathbf{y} = \mathbf{r} * \mathbf{c}_{\text{CMA}b}. \quad (7.5)$$

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

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2 \quad (7.6)$$

where

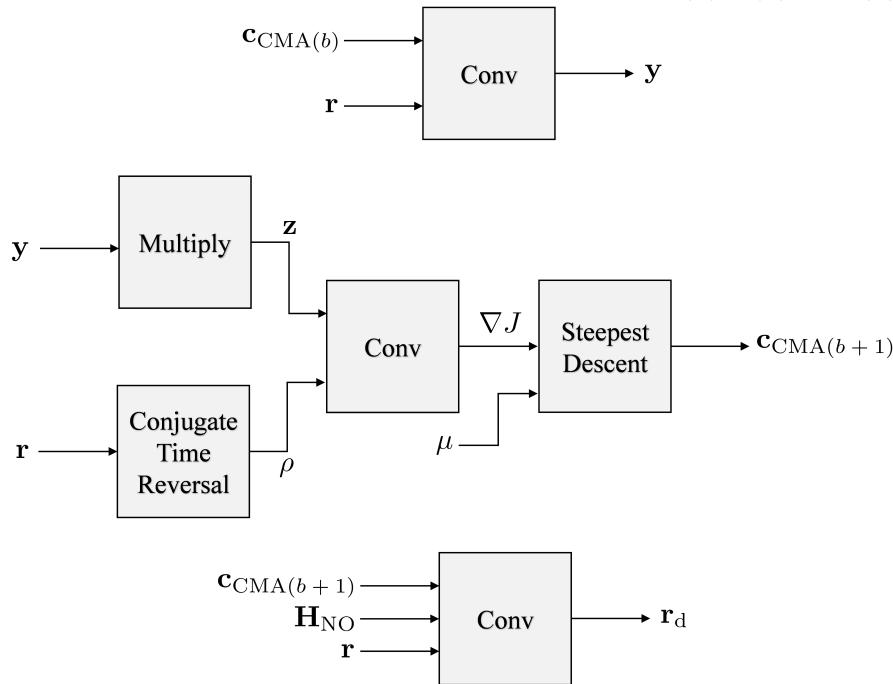
$$\begin{aligned} b(n) &= \sum_{m=0}^{L_{pkt}-1} z(m)\rho(n-m) \\ &= \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-n) \end{aligned} \quad (7.7)$$

using convolution. Once ∇J is calculated, the steepest decent algorithm is applied

$$\mathbf{c}_{\text{CMA}(b+1)} = \mathbf{c}_{\text{CMA}(b)} - \mu \nabla J. \quad (7.8)$$

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



In Section 6.2.2, the direct application of CMA was massaged into convolution. Figure 7.5 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 6.27

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-k), \quad -L_1 \leq k \leq L_2. \quad (7.9)$$

Long summations are slow in GPUs. Table 7.3 lists the comparison on computing $\nabla J(k)$ verse using convolution.

Table 7.2: Defining start and stop lines for timing comparison in Listing 5.1.

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

Table 7.3: Defining start and stop lines for timing comparison in Listing 5.1.

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

7.3 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 6.32 and is implemented directly in the GPU. To save execution time, the FFT of the detection filter H_{NO} multiplied at the same time FDE1 is calculated

$$R_{d1}(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})H_{NO}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1 \quad (7.10)$$

$$R_{d2}(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})H_{NO}(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1 \quad (7.11)$$

where $R(e^{j\omega_k})$ and $R_d(e^{j\omega_k})$ is the FFT r and r_d at ω_k . Figures 7.6 and 7.7 show block diagrams of how FDE1 and FDE2 are implemented in the GPUs.

Figure 7.6: Diagram showing Frequency Domain Equalizer One is implemented in the frequency domain in GPUs.

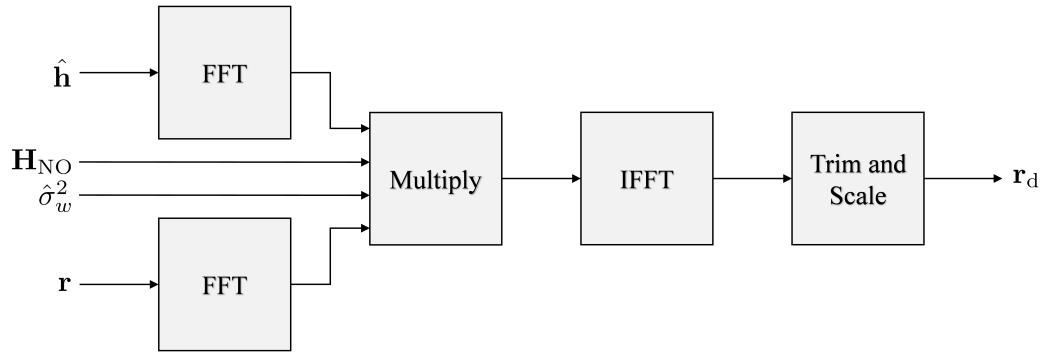
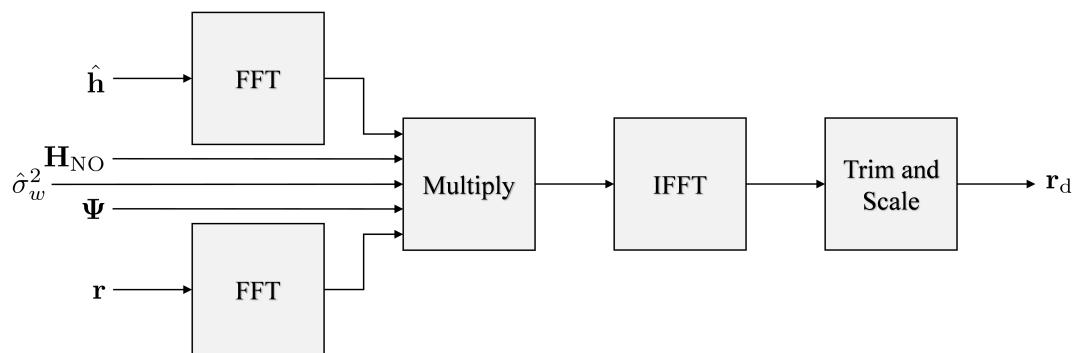


Figure 7.7: Diagram showing Frequency Domain Equalizer Two is implemented in the frequency domain in GPUs.



Chapter 8

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. 7
- [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. 8
- [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. 9
- [4] E. Perrins, “FEC systems for aeronautical telemetry,” **IEEE Transactions on Aerospace and Electronic Systems**, vol. 49, no. 4, pp. 2340–2352, October 2013. 9, 27
- [5] Wikipedia, “Graphics processing unit,” 2015. [Online]. Available: http://en.wikipedia.org/wiki/Graphics_processing_unit 11
- [6] NVIDIA, “Cuda toolkit documentation,” 2017. [Online]. Available: <http://docs.nvidia.com/cuda/> 11, 69
- [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. 25
- [8] Wikipedia, “Fastest fourier transform in the west,” 2017. [Online]. Available: <http://www.fftw.org/> 29
- [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. 29
- [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. 56, 58, 60
- [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. 64

- [12] Wikipedia, “Sparse matrix,” 2017. [Online]. Available: https://en.wikipedia.org/wiki/Sparse_matrix 69
- [13] M. Hayes, **Statistical Digital Signal Processing and Modeling**. New York: John Wiley & Sons, 1996. 69