

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 Signal Processing with GPUs	5
3.1 Simple GPU code example	5
3.2 GPU kernel using threads and thread blocks	8
3.3 GPU memory	9
3.4 Cuda Libraries	12
3.5 Cuda Convolution	12
3.6 Thread Optimization	17
3.7 CPU GPU Pipelining	20
4 System Overview	29
4.1 Overview	29
4.2 Preamble Detection	30
4.3 Frequency Offset Compensation	32
4.4 Channel Estimation	33

4.5	Noise Variance Estimation	33
4.6	Symbol-by-Symbol Detector	33
5	Equalizer Equations	35
5.1	Overview	35
5.2	Zero-Forcing and Minimum Mean Square Error Equalizers	35
5.2.1	Zero-Forcing	36
5.2.2	The Constant Modulus Algorithm	40
5.2.3	The Frequency Domain Equalizers	44
6	Equalizer GPU Implementation	47
6.1	CUDA Batched Processing	47
6.2	Batched Convolution	48
6.3	Equalizer Implementations	49
6.4	Zero-Forcing and MMSE GPU Implementation	50
6.5	Constant Modulus Algorithm GPU Implementation	51
6.6	Frequency Domain Equalizer One and Two GPU Implementation	53
7	Equalizer Performance	55
8	Final Summary	57
Bibliography		58

List of Tables

3.1	The computational resources available with three NVIDIA GPUs used in this thesis (1x Tesla K40c 2x Tesla K20c).	10
3.2	Defining start and stop lines for timing comparison in Listing 3.4.	14
3.3	Convolution computation times with signal length $2^{15} = 32768$ and filter length 186 on a Tesla K40c GPU.	15
3.4	Convolution computation times with signal length 12672 and filter length 186 on a Tesla K40c GPU.	15

List of Figures

3.1	A block diagram of how a CPU sequentially performs vector addition.	6
3.2	A block diagram of how a GPU performs vector addition in parallel.	6
3.3	Block 0 32 threads launched in 4 thread blocks with 8 threads per block.	9
3.4	36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.	9
3.5	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.	10
3.6	NVIDIA Tesla K40c and K20c.	11
3.7	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.	11
3.8	Comparison of complex convolution on CPU to GPU with varying signal lengths without lower bounding.	15
3.9	Comparison of complex convolution on CPU to GPU with varying signal lengths with lower bounding.	16
3.10	Plot showing when CPU convolution is faster than GPU convolution.	17
3.11	Plot showing trade offs with convolution in GPUs.	18
3.12	Lower bounded plot showing trade offs with convolution in GPUs.	19
3.13	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
3.14	ConvGPU thread optimization 128 threads per block 0.006811.	21

3.15	The typical approach of CPU and GPU operations. This block diagram shows a Profile of Listing 3.2.	21
3.16	GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 3.3.	22
3.17	A block diagram of pipelining a CPU with three GPUs.	23
4.1	A block diagram of the estimation process.	29
4.2	A block diagram of the equalization and symbol detector process.	30
4.3	The iNET packet structure.	30
4.4	Offset Quadrature Phase Shift Keying symbol by symbol detector.	34
5.1	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	43
5.2	I need help on this one!!!!	45
6.1	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	48
6.2	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	48
6.3	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	51
6.4	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	52
6.5	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	53
6.6	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	54
6.7	Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$	54

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

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 [1].

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 [2].

3.1 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

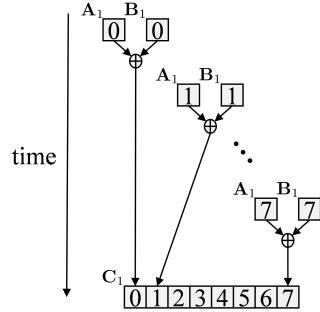


Figure 3.1: A block diagram of how a CPU sequentially performs vector addition.

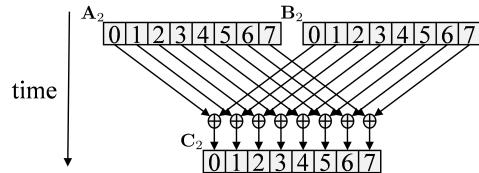


Figure 3.2: A block diagram of how a GPU performs vector addition in parallel.

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

Listing 3.1 shows a simple program that sums two vectors together

$$\begin{aligned} C_1 &= A_1 + B_1 \\ C_2 &= A_2 + B_2 \end{aligned} \tag{3.1}$$

where each vector is length 1024. Figure 3.1 shows how the CPU computes C_1 by summing elements of A_1 and B_1 together *sequentially*. Figure 3.2 shows how the GPU computes C_2 by summing elements of A_2 and B_2 together *in parallel*. The GPU kernel computes every element of C_2 in parallel while the CPU computes one element of C_1 at a time.

Listing 3.1: Comparison of CPU verse GPU code.

```

1 #include <iostream>
2 #include <stdlib.h>
3 #include <math.h>
4 using namespace std;
5
6 void VecAddCPU(float* destination, float* source0, float* source1, int myLength) {
7     for(int i = 0; i < myLength; i++)
8         destination[i] = source0[i] + source1[i];
9 }
10
11 __global__ void VecAddGPU(float* destination, float* source0, float* source1, int lastThread) {
12     int i = blockIdx.x*blockDim.x + threadIdx.x;
13
14     // don't access elements out of bounds
15     if(i >= lastThread)
16         return;
17
18     destination[i] = source0[i] + source1[i];
19 }
20
21
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 }

```

3.2 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 $\text{blockDim} = \text{numThreadsPerBlock}$. 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

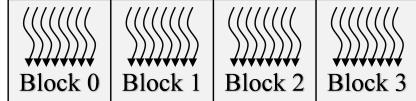


Figure 3.3: Block 0 32 threads launched in 4 thread blocks with 8 threads per block.

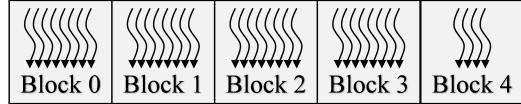


Figure 3.4: 36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.

integer multiple of T . Figure 3.3 shows 32 threads launched in 4 thread blocks with 8 threads per block. Figure 3.4 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.

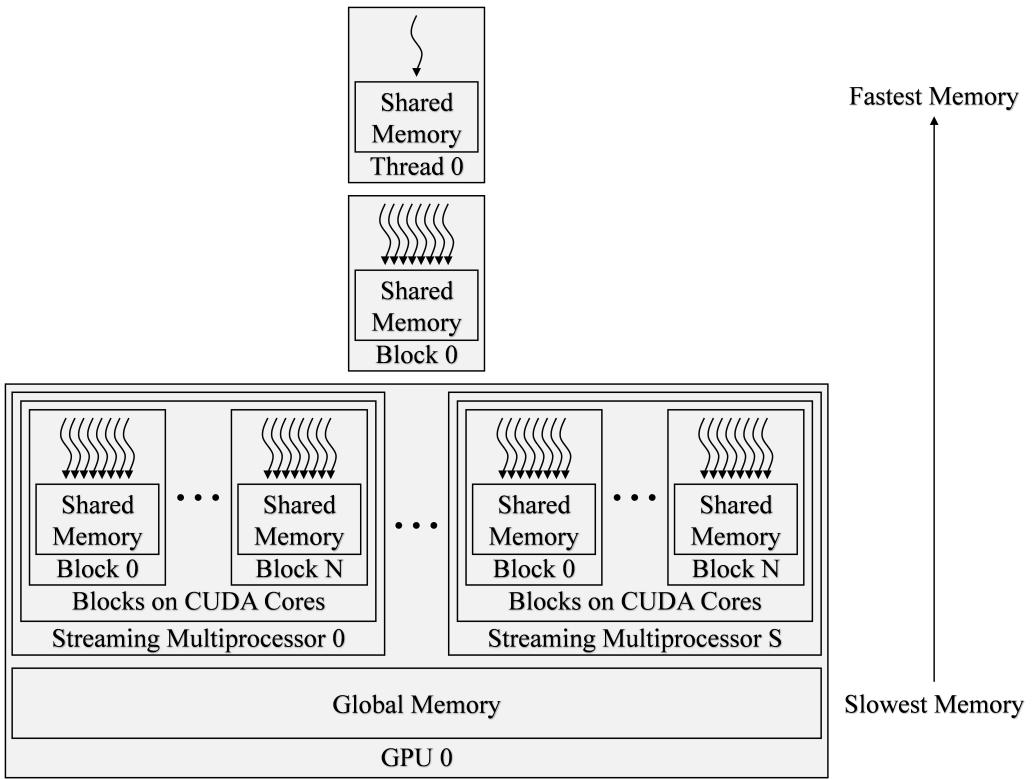
3.3 GPU memory

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 3.1 gives some specifications and Figure 3.6 shows the form factor of the these GPUs. The red box in Figure 3.7 show the GPU

Figure 3.5: 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 3.1: 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 3.6: NVIDIA Tesla K40c and K20c.

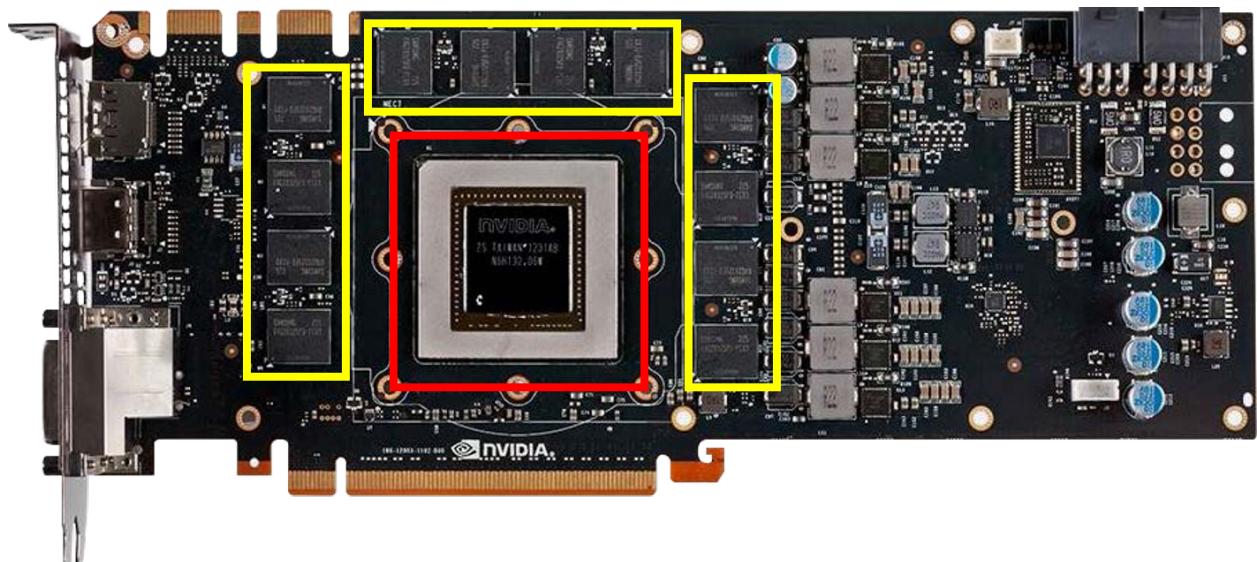


Figure 3.7: Example of an NVIDIA GPU card. The SRAM is shown to be boxed in yellow. The GPU chip is shown to be boxed in red.

3.4 Cuda Libraries

CUDA isn't just a programming language, it also has many libraries that are extremely useful that are optimized for NVIDIA GPUs. 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 libraries for anything that might be useful.

Some libraries used in this Thesis are

- cufft
- cublas
- cusolver
- cusolverSp

3.5 Cuda Convolution

An important tool to Digital Communications and Digital Signal Processing is convolution.

Time domain convolution is

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

where $x(m)$, $h(n - m)$ and $y(n)$ are complex. While Listing 3.1 is a simple and good example showing how to program GPUs, frankly, it is pretty boring and doesn't display the real challenges and tradeoffs of GPUs. Listing 3.4 shows four different ways of implementing convolution

- time 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 (6.5) in ConvCPU directly from line 6 to 30. The GPU implements time domain convolution using global memory in the GPU kernel ConvGPU. ConvGPU on lines 32 to 59 is a parallel of ConvCPU. ConvGPU implements time domain convolution by accessing global memory for any needed element of the signal and filter.

Threads accessing the same elements of the filter in global memory seems like a waste of valuable clock cycles. The GPU kernel ConvGPUs shared on lines 61 to 96 differs slightly from ConvGPU by making L_h threads save the filter to shared memory from global memory.

Any graduate of a signal processing class knows the trade off of convolution in the time versus frequency domain. Time domain convolution takes $\approx L_s L_h$ complex multiplies where L_s is the signal length and L_h is the filter length. Frequency domain convolution takes $\approx \frac{3N_{FFT}}{2} \log_2(N_{FFT}) + N_{FFT}$ complex multiplies where N_{FFT} is the next multiple of 2 above the convolution length $L_s + L_h - 1$.

Lines 213 to 232 show how to do frequency domain convolution using the cuFFT library and two simple GPU kernels.

So the questions are: When should I stay on the host and run my convolution on the CPU? When should I copy my vectors to the GPU, run my convolution, then copy the vectors back to the host? If I am going to run my convolution the GPU, when should I only use global memory? When should I use shared memory? When should I do convolution in the frequency domain?

The answer to all of the questions is...it depends on your signal length, filter length, CPU, GPU and memory.

A CUDA programmer can make an educated guess on which way may be faster, but until the algorithms have been implemented and timed, there is no definite answer. Figures 3.8 to ?? compare the four different ways of doing convolution implemented in Listing 3.1 on a Tesla K40c GPU. The filter length for Figures 3.8 to ?? is 186. The filter length for Figure 3.12 is 10. Table 3.2 lists how the kernels were timed. Note that all memory transfers from host to device and device to host were included in execution times.

Table 3.2: Defining start and stop lines for timing comparison in Listing 3.4.

Algorithm	Start Timing Line	Stop Timing Line
CPU time domain (ConvCPU)	182	183
GPU time domain global (ConvGPU)	187	195
GPU time domain shared (ConvGPUshared)	200	208
GPU frequency domain	212	231

Figure 3.8 shows the raw computation time on the K40c for each algorithm in Table 3.2 with the filter length set to 186. The figure has many irregularities because the operating system doesn't always provide as much resources as needed.

Figure 3.9 is generated from the raw computation data but lower bounded by searching for a minimum every 20 samples. Judging my the Figure, for long signals convolved with 186 tap filters, GPU convolution is much faster than CPU.

Looking at the bottom left of Figures 3.8 and 3.9, a DSP engineer would ask, “Is it ever better to do convolution on the CPU rather than GPU?” Yes, but for very short signals with a 186 tap filter. Figure 3.10 shows when a CPU is faster than a GPU.

Now for the interesting question, if I am going to do convolution in the GPU, should I do time domain or frequency domain convolution? If I do time domain convolution, should I use global or shared memory? Figure 3.11 shows that the answer: “It depends.” A good choice is to do frequency domain, but not always. As the signal length increases, the frequency domain execution time has a staircase shape because the signal is zero padded out to the next power of 2 to leverage the Cooley-Tukey FFT algorithm [?].

Usually, when implementing convolution, the signal and filter length is set. It is good practice to implement convolution in the CPU and GPU in many ways to evaluate which way is best. Choosing the signal length to be 2^{15} and the filter length to be 186, Tables 3.3 and 3.4 shows how the different algorithms compare for different signal lengths.

Figure 3.8: Comparison of complex convolution on CPU to GPU with varying signal lengths without lower bounding.

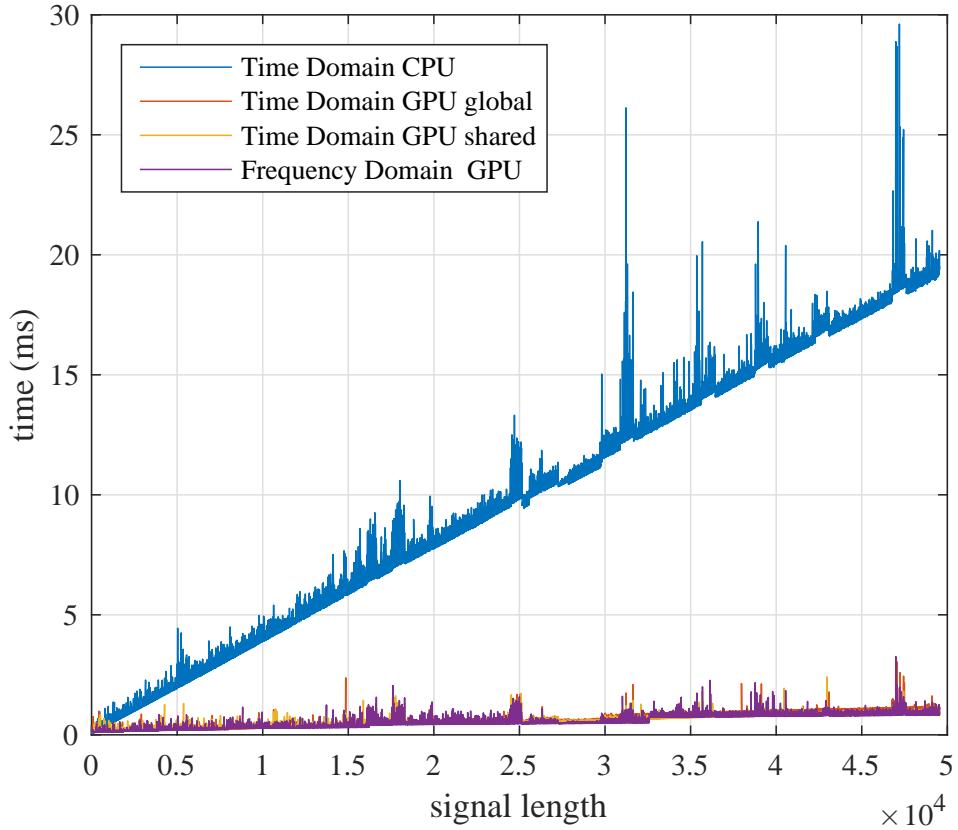


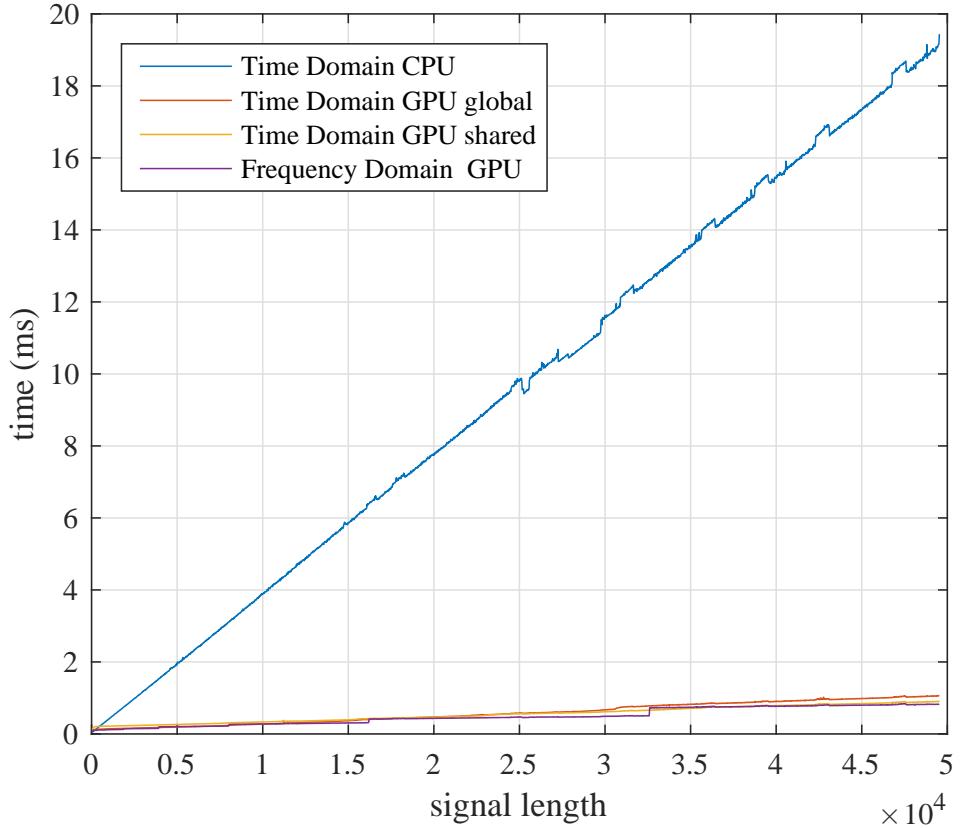
Table 3.3: Convolution computation times with signal length $2^{15} = 32768$ and filter length 186 on a Tesla K40c GPU.

Algorithm	time (ms)
CPU time domain (ConvCPU)	131.943
GPU time domain global (ConvGPU)	7.79477
GPU time domain shared (ConvGPUshared)	6.77736
GPU frequency domain	5.81085

Table 3.4: Convolution computation times with signal length 12672 and filter length 186 on a Tesla K40c GPU.

Algorithm	time (ms)
CPU time domain (ConvCPU)	5.04636
GPU time domain global (ConvGPU)	0.320013
GPU time domain shared (ConvGPUshared)	0.292616
GPU frequency domain	0.284187

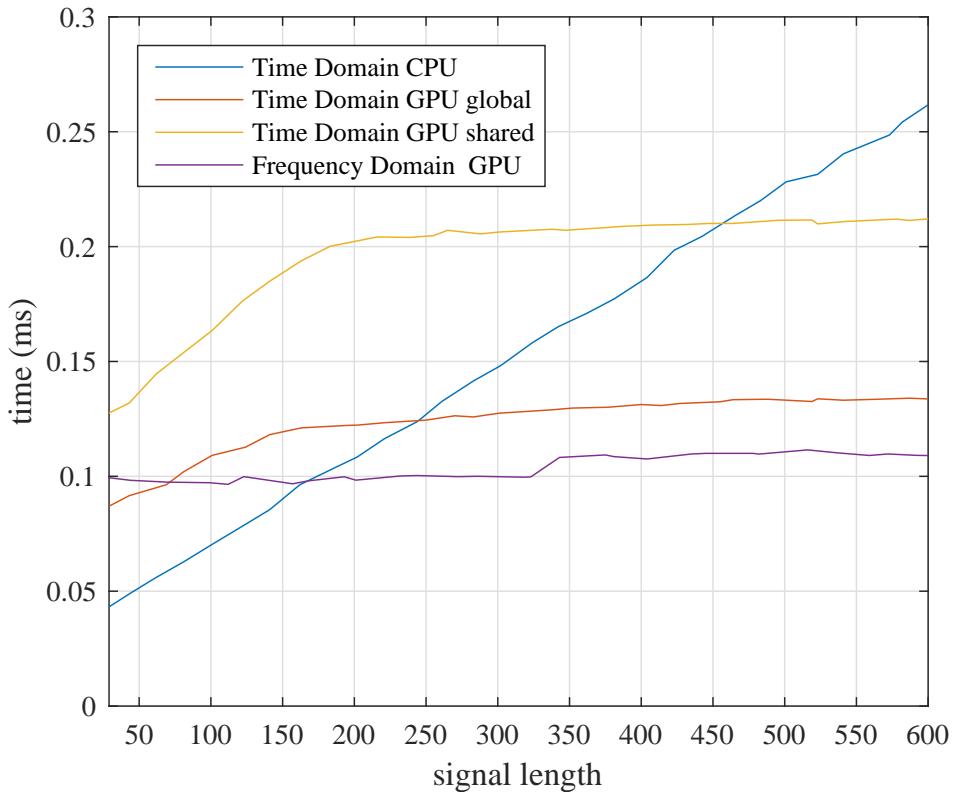
Figure 3.9: Comparison of complex convolution on CPU to GPU with varying signal lengths with lower bounding.



If the length of the filter were changed from 186 to 10 taps, everything changes. Figure 3.12 shows execution time for a varying length complex signal convolved with a 10 tap complex filter. Contrary to the text book number of flops comparison, convolution in the frequency domain is never faster than time domain in GPUs for a 10 tap filter, even with an extremely long signal. Time domain convolution using shared memory in ConvGPUshared is fastest for a signal of significant length. ConvGPUshared performs very well because only 10 threads per thread block need to access global memory for the filter coefficients while all other threads only need to access global memory for the signal.

Long story short, most of the time a GPU DSP engineer is given a set length of signal and filter for convolution. As Figures 3.8 through 3.11 have shown, unless every implementation is explored, there is no way of saying which implementation is fastest.

Figure 3.10: Plot showing when CPU convolution is faster than GPU convolution.

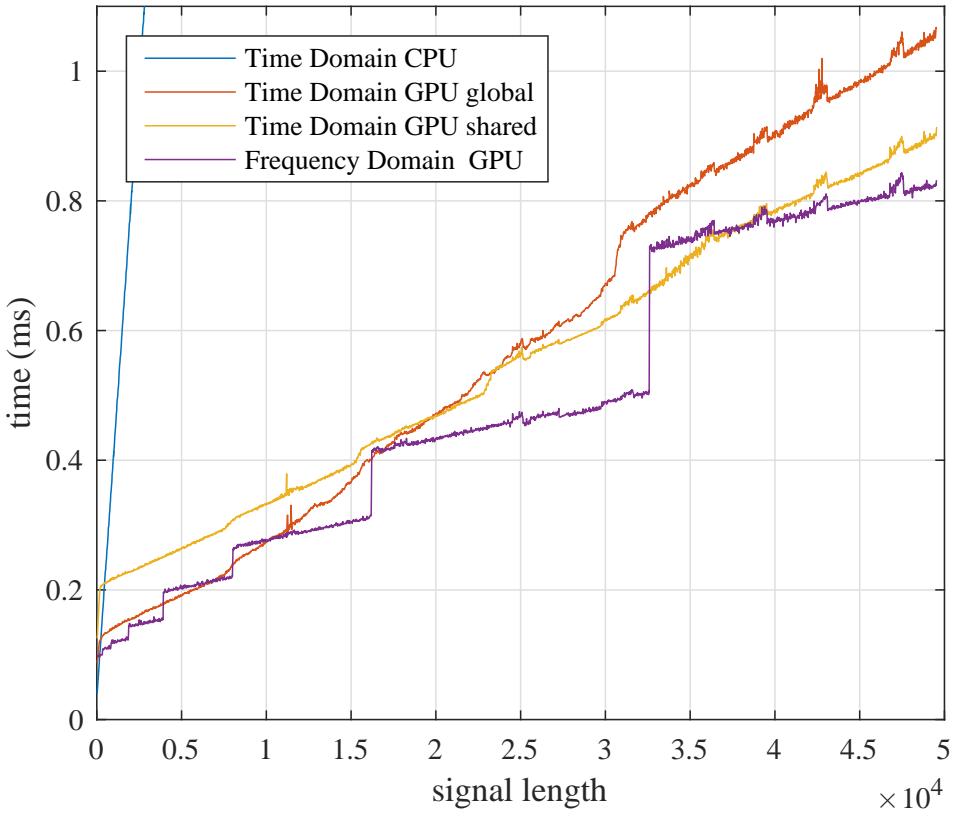


3.6 Thread Optimization

When first introduced to GPUs, a programmer might be tempted to launch as many threads per block as possible. But, notice in Listing 3.4 lines 190, 203, 219 and 225 launch a minimum of 96 threads per block and maximum of 192 threads per block. Running the GPU with less threads per block with lower occupancy leaves each thread with more resources and memory bandwidth. Running the GPU with more threads per block with higher occupancy utilizes the full parallel might of the GPU. If 1024 threads per block were always launched, lighter computation GPU kernels will be starving for memory while other thread blocks eat up the memory bandwidth. If 32 threads per block were always launched, heavier computation GPU kernels will be swimming in resources.

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.

Figure 3.11: Plot showing trade offs with convolution in GPUs.

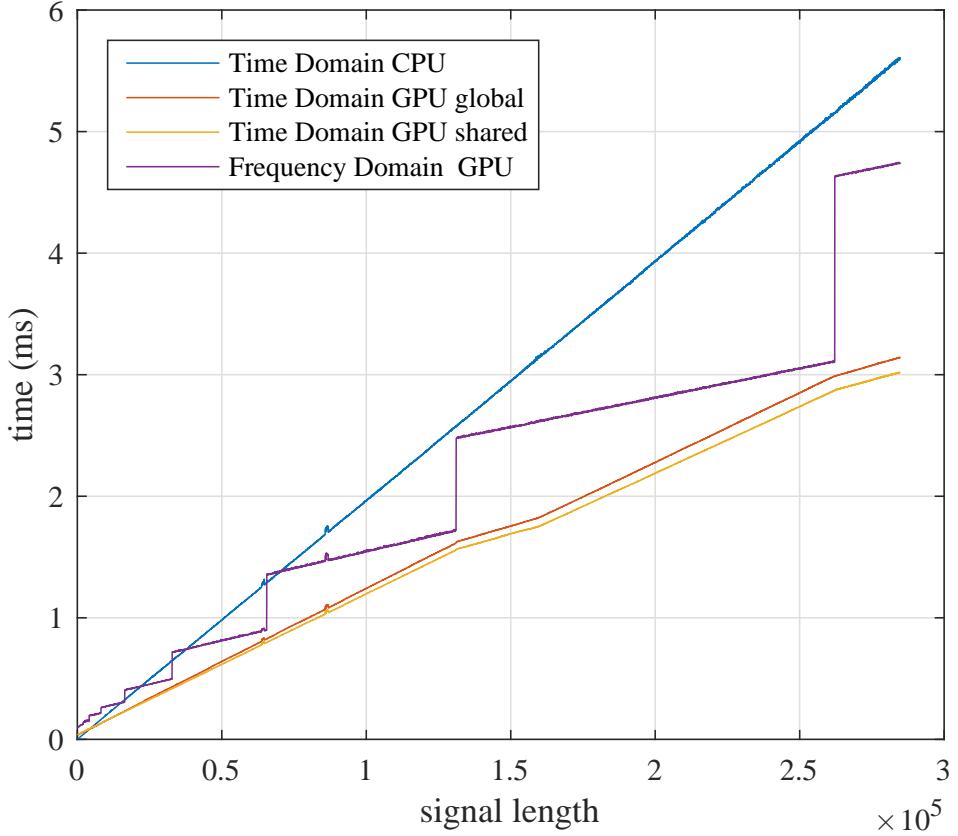


Knowing the perfect number of threads per block to launch is challenging to calculate. Luckily, there is a finite number of possible threads per block, 1 to 1024. A simple test program could time a GPU kernel while sweeping the number of threads per block from 1 to 1024. The number of threads per block 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 3.13 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

Figure 3.12: Lower bounded plot showing trade offs with convolution in GPUs.

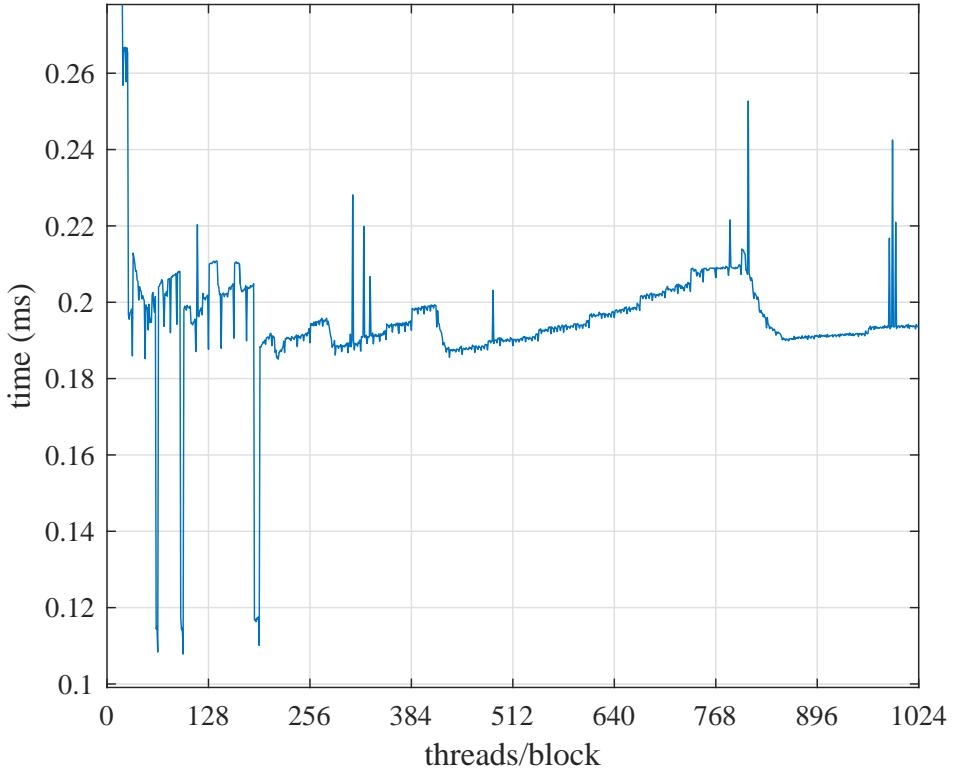


96 threads per block only transfer 96 filter coefficients to shared memory from global memory. Luckily, launching 192 threads per block is near optimal with an execution time of 0.1101ms. By simply adjusting the number of threads per block, ConvGPUshared can have a $2\times$ speed up.

Adjusting the number of threads per block doesn't always drastically speed up GPU kernels. Figure 3.13 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.”

Figure 3.13: 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.



3.7 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?

Listing 3.2 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 3.15 shows a block diagram of what is happening on the CPU and GPU in Listing 3.2. 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 3.14: ConvGPU thread optimization 128 threads per block 0.006811.

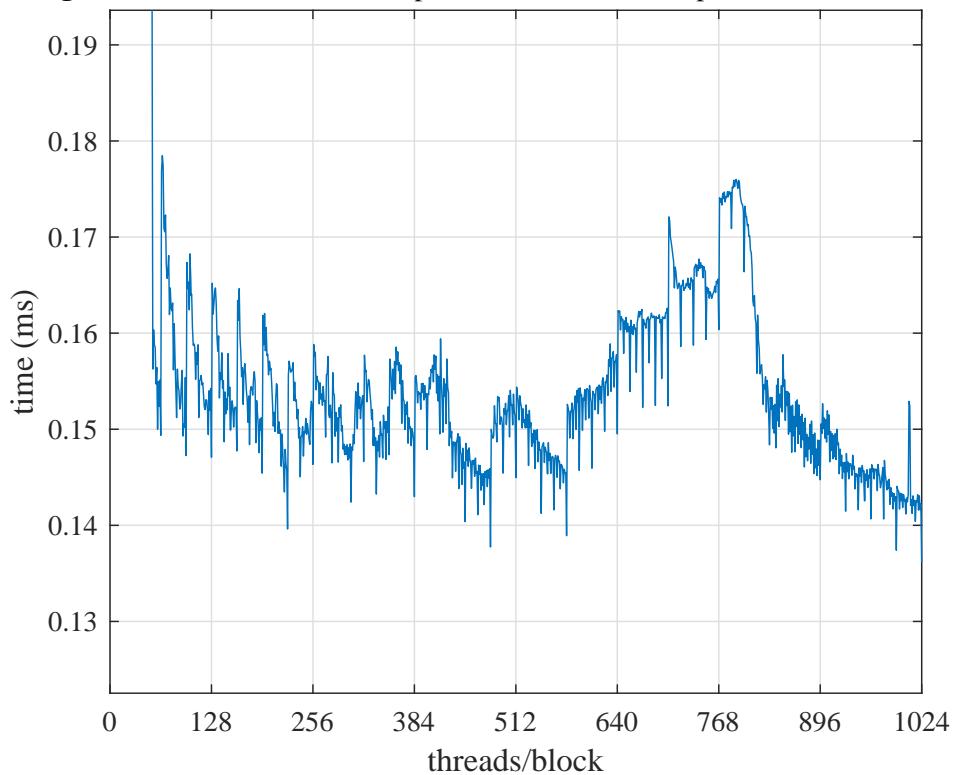
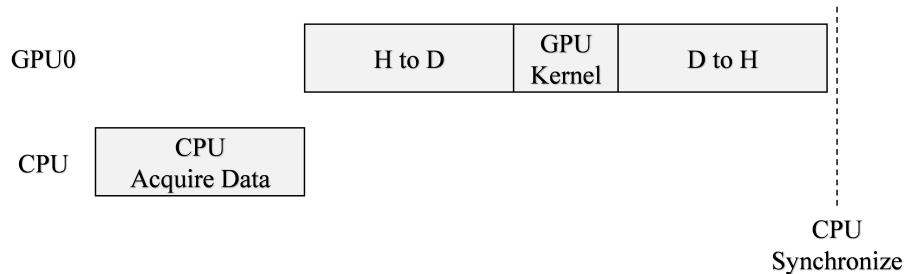


Figure 3.15: The typical approach of CPU and GPU operations. This block diagram shows a Profile of Listing 3.2.



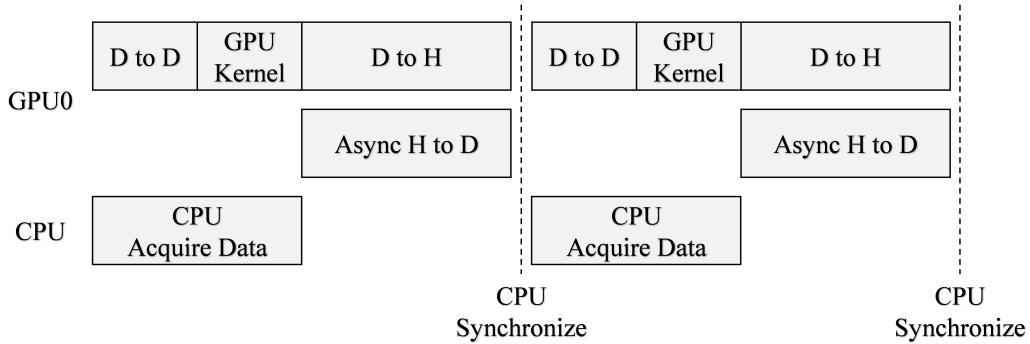
Listing 3.2: Example code Simple example of the CPU acquiring data from myADC, copying from host to device, processing data on the device then copying from device to host. No processing occurs on device while CPU is acquiring data.

```

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);

```

Figure 3.16: GPU and CPU operations can be pipelined. This block diagram shows a Profile of Listing 3.3.



```

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 3.3 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.

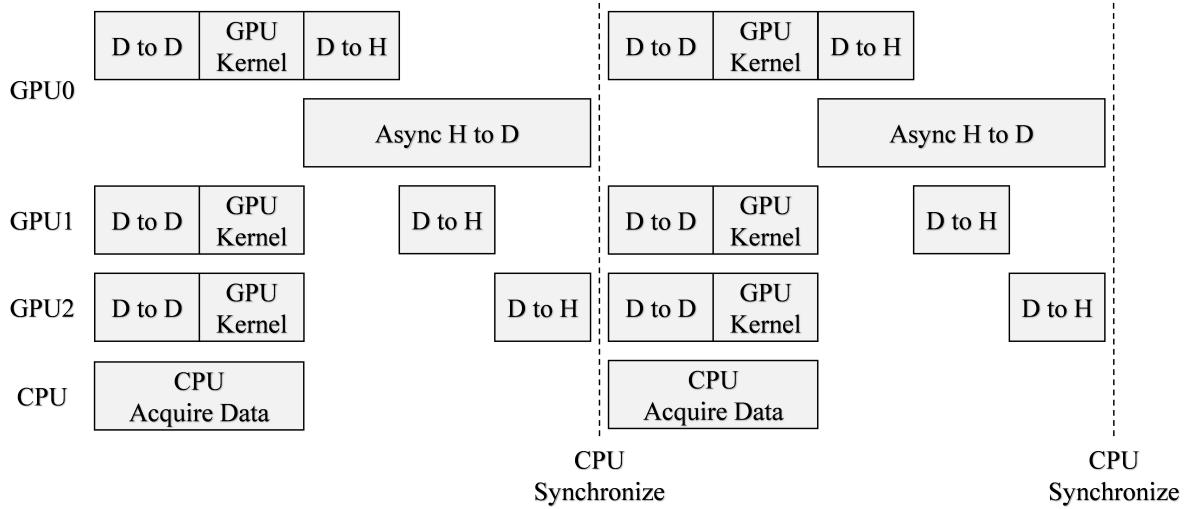
Listing 3.3: Example code Simple of the CPU acquiring data from myADC, copying from host to device, processing data on the device then copying from device to host. No processing occurs on device while CPU is acquiring data.

```

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 Acquire Data
10    myADC.acquire(vec);
11    cudaMemcpyAsync(dev_temp, vec, numBytes, cudaMemcpyHostToDevice);
12

```

Figure 3.17: A block diagram of pipelining a CPU with three GPUs.



```

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 Acquire 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 3.17 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.

Listing 3.4: CUDA code to performing complex convolution four different ways: time domain CPU, time domain GPU, time domain GPU using shared memory and frequency domain GPU.

```

1 #include <cufft.h>
2 using namespace std;
3 // Length of Filter (186 is a magic number)
4 const int LH = 186;
5
6 void ConvCPU(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
7     for(int yIdx = 0; yIdx < Lx+Lh-1; yIdx++) {
8         cufftComplex temp;
9         temp.x = 0;
10        temp.y = 0;
11        for(int hIdx = 0; hIdx < Lh; hIdx++) {
12            int xAccessIdx = yIdx-hIdx;
13            if(xAccessIdx>=0 && xAccessIdx<Lx) {
14                // temp += x[xAccessIdx]*h[hIdx];
15                // (A+jB) (C+jD) = (AC-BD) + j(AD+BC)
16                float A = x[xAccessIdx].x;
17                float B = x[xAccessIdx].y;
18                float C = h[hIdx].x;
19                float D = h[hIdx].y;
20                cufftComplex complexMult;
21                complexMult.x = A*C-B*D;
22                complexMult.y = A*D+B*C;
23
24                temp.x += complexMult.x;
25                temp.y += complexMult.y;
26            }
27        }
28        y[yIdx] = temp;
29    }
30 }
31
32 __global__ void ConvGPU(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh) {
33     int yIdx = blockIdx.x*blockDim.x + threadIdx.x;
34     int lastThread = Lx+Lh-1;
35     // Don't access elements out of bounds
36     if(yIdx >= lastThread)
37         return;
38     cufftComplex temp;
39     temp.x = 0;
40     temp.y = 0;
41     for(int hIdx = 0; hIdx < Lh; hIdx++) {
42         int xAccessIdx = yIdx-hIdx;
43         if(xAccessIdx>=0 && xAccessIdx<Lx) {
44             // temp += x[xAccessIdx]*h[hIdx];
45             // (A+jB) (C+jD) = (AC-BD) + j(AD+BC)
46             float A = x[xAccessIdx].x;
47             float B = x[xAccessIdx].y;
48             float C = h[hIdx].x;
49             float D = h[hIdx].y;
50             cufftComplex complexMult;
51             complexMult.x = A*C-B*D;
52             complexMult.y = A*D+B*C;
53
54             temp.x += complexMult.x;
55             temp.y += complexMult.y;
56         }
57         y[yIdx] = temp;
58     }
59 }
60
61 __global__ void ConvGPUsShared(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh) {
62     int yIdx = blockIdx.x*blockDim.x + threadIdx.x;
63     // Be sure to read in full h_shared before checking lastThread
64     // First Lh threads in thread block read h from global memory into shared memory
65     __shared__ cufftComplex h_shared[LH];

```

```

66     if(threadIdx.x < LH)
67         h_shared[threadIdx.x] = h[threadIdx.x];
68     int lastThread = Lx+Lh-1;
69     // Don't access elements out of bounds
70     if(yIdx >= lastThread)
71         return;
72     // Thread barrier, ensures threads wait until h has been read into h_shared
73     __syncthreads();
74     cufftComplex temp;
75     temp.x = 0;
76     temp.y = 0;
77     for(int hIdx = 0; hIdx < Lh; hIdx++){
78         int xAccessIdx = yIdx-hIdx;
79         if(xAccessIdx>=0 && xAccessIdx<Lx){
80             // temp += x[xAccessIdx]*h_shared[hIdx];
81
82             // (A+jB) (C+jD) = (AC-BD) + j(AD+BC)
83             float A = x[xAccessIdx].x;
84             float B = x[xAccessIdx].y;
85             float C = h_shared[hIdx].x;
86             float D = h_shared[hIdx].y;
87             cufftComplex complexMult;
88             complexMult.x = A*C-B*D;
89             complexMult.y = A*D+B*C;
90
91             temp.x += complexMult.x;
92             temp.y += complexMult.y;
93         }
94         y[yIdx] = temp;
95     }
96 }
97
98 __global__ void PointToPointMultiply(cufftComplex* vec0, cufftComplex* vec1, int lastThread){
99     int i = blockIdx.x*blockDim.x + threadIdx.x;
100    // Don't access elements out of bounds
101    if(i >= lastThread)
102        return;
103    // vec0[i] = vec0[i]*vec1[i];
104    // (A+jB) (C+jD) = (AC-BD) + j(AD+BC)
105    float A = vec0[i].x;
106    float B = vec0[i].y;
107    float C = vec1[i].x;
108    float D = vec1[i].y;
109    cufftComplex complexMult;
110    complexMult.x = A*C-B*D;
111    complexMult.y = A*D+B*C;
112    vec0[i] = complexMult;
113 }
114
115 __global__ void ScalarMultiply(cufftComplex* vec0, float scalar, int lastThread){
116     int i = blockIdx.x*blockDim.x + threadIdx.x;
117     // Don't access elements out of bounds
118     if(i >= lastThread)
119         return;
120     cufftComplex scalarMult;
121     scalarMult.x = vec0[i].x*scalar;
122     scalarMult.y = vec0[i].y*scalar;
123     vec0[i] = scalarMult;
124 }
125
126 int main(){
127     int mySignalLength = pow(2,15);
128     int myFilterLength = LH;
129     int myConvLength    = mySignalLength + myFilterLength - 1;
130     int Nfft            = pow(2, ceil(log(myConvLength)/log(2)));
131     int numThreadsPerBlock;
132     int numBlocks;
133 }
```

```

134     cufftHandle plan;
135     int n[1] = {Nfft};
136     cufftPlanMany(&plan, 1, n, NULL, 1, 1, NULL, 1, 1, CUFFT_C2C, 1);
137
138     // Allocate memory on host
139     cufftComplex *mySignal1;
140     cufftComplex *myFilter1;
141     cufftComplex *myConv1;
142     cufftComplex *myConv2;
143     cufftComplex *myConv3;
144     cufftComplex *myConv4;
145     mySignal1      = (cufftComplex*) malloc(mySignalLength*sizeof(cufftComplex));
146     myFilter1      = (cufftComplex*) malloc(myFilterLength*sizeof(cufftComplex));
147     myConv1        = (cufftComplex*) malloc(myConvLength  * sizeof(cufftComplex));
148     myConv2        = (cufftComplex*) malloc(myConvLength  * sizeof(cufftComplex));
149     myConv3        = (cufftComplex*) malloc(myConvLength  * sizeof(cufftComplex));
150     myConv4        = (cufftComplex*) malloc(Nfft           * sizeof(cufftComplex));
151     for(int i = 0; i < mySignalLength; i++){
152         mySignal1[i].x = rand()%100-50;
153         mySignal1[i].y = rand()%100-50;
154     }
155     for(int i = 0; i < myFilterLength; i++){
156         myFilter1[i].x = rand()%100-50;
157         myFilter1[i].y = rand()%100-50;
158     }
159
160     // Allocate memory on device
161     cufftComplex *dev_mySignal2;
162     cufftComplex *dev_myFilter2;
163     cufftComplex *dev_myConv2;
164     cufftComplex *dev_mySignal3;
165     cufftComplex *dev_myFilter3;
166     cufftComplex *dev_myConv3;
167     cufftComplex *dev_mySignal4;
168     cufftComplex *dev_myFilter4;
169     cufftComplex *dev_myConv4;
170     cudaMalloc(&dev_mySignal2, mySignalLength*sizeof(cufftComplex));
171     cudaMalloc(&dev_myFilter2, myFilterLength*sizeof(cufftComplex));
172     cudaMalloc(&dev_myConv2,   myConvLength  * sizeof(cufftComplex));
173     cudaMalloc(&dev_mySignal3, mySignalLength*sizeof(cufftComplex));
174     cudaMalloc(&dev_myFilter3, myFilterLength*sizeof(cufftComplex));
175     cudaMalloc(&dev_myConv3,   myConvLength  * sizeof(cufftComplex));
176     cudaMalloc(&dev_mySignal4, Nfft           * sizeof(cufftComplex));
177     cudaMalloc(&dev_myFilter4, Nfft           * sizeof(cufftComplex));
178     cudaMalloc(&dev_myConv4,   Nfft           * sizeof(cufftComplex));
179
180     /*****
181             CPU Time Domain Direct Convolution
182     *****/
183     ConvCPU(myConv1,mySignal1,myFilter1,mySignalLength,myFilterLength);
184
185     /*****
186             GPU Time DomainDirect Convolution
187     *****/
188     cudaMemcpy(dev_mySignal2, mySignal1, sizeof(cufftComplex)*mySignalLength,
189               cudaMemcpyHostToDevice);
189     cudaMemcpy(dev_myFilter2, myFilter1, sizeof(cufftComplex)*myFilterLength,
190               cudaMemcpyHostToDevice);
190     numThreadsPerBlock = 128;
191     numBlocks = myConvLength/numThreadsPerBlock;
192     if(myConvLength % numThreadsPerBlock > 0)
193         numBlocks++;
194     ConvGPU<<<numBlocks, numThreadsPerBlock>>>(dev_myConv2, dev_mySignal2, dev_myFilter2,
195                                                 mySignalLength, myFilterLength);
196     cudaMemcpy(myConv2, dev_myConv2, myConvLength*sizeof(cufftComplex),
197               cudaMemcpyDeviceToHost);
198

```

```

198     /-----  

199             GPU Time Domain Convolution Using Shared Memory  

200     -----*/  

201     cudaMemcpy(dev_mySignal3, mySignall, sizeof(cufftComplex)*mySignalLength,  

202                 cudaMemcpyHostToDevice);  

203     cudaMemcpy(dev_myFilter3, myFilterl, sizeof(cufftComplex)*myFilterLength,  

204                 cudaMemcpyHostToDevice);  

205     numTreadsPerBlock = 192;  

206     numBlocks = myConvLength/numTreadsPerBlock;  

207     if(myConvLength % numTreadsPerBlock > 0)  

208         numBlocks++;  

209     ConvGPUshared<<<numBlocks, numTreadsPerBlock>>>(dev_myConv3, dev_mySignal3, dev_myFilter3  

210                 , mySignalLength, myFilterLength);  

211     cudaMemcpy(myConv3, dev_myConv3, myConvLength*sizeof(cufftComplex),  

212                 cudaMemcpyDeviceToHost);  

213  

214     /-----  

215             GPU Frequency Domain Convolution using cuFFT  

216     -----*/  

217     cudaMemset(dev_mySignal4, 0, Nfft*sizeof(cufftComplex));  

218     cudaMemset(dev_myFilter4, 0, Nfft*sizeof(cufftComplex));  

219     cudaMemcpy(dev_mySignal4, mySignall, sizeof(cufftComplex)*mySignalLength,  

220                 cudaMemcpyHostToDevice);  

221     cudaMemcpy(dev_myFilter4, myFilterl, sizeof(cufftComplex)*myFilterLength,  

222                 cudaMemcpyHostToDevice);  

223     cufftExecC2C(plan, dev_mySignal4, dev_mySignal4, CUFFT_FORWARD);  

224     cufftExecC2C(plan, dev_myFilter4, dev_myFilter4, CUFFT_FORWARD);  

225     numTreadsPerBlock = 96;  

226     numBlocks = Nfft/numTreadsPerBlock;  

227     if(Nfft % numTreadsPerBlock > 0)  

228         numBlocks++;  

229     PointToPointMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal4, dev_myFilter4, Nfft  

230                 );  

231     cufftExecC2C(plan, dev_mySignal4, dev_mySignal4, CUFFT_INVERSE);  

232     numTreadsPerBlock = 128;  

233     numBlocks = Nfft/numTreadsPerBlock;  

234     if(Nfft % numTreadsPerBlock > 0)  

235         numBlocks++;  

236     float scalar = 1.0/((float)Nfft);  

237     ScalarMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal4, scalar, Nfft);  

238     cudaMemcpy(myConv4, dev_mySignal4, myConvLength*sizeof(cufftComplex),  

239                 cudaMemcpyDeviceToHost);  

240     cufftDestroy(plan);  

241  

242     // Free vectors on CPU  

243     free(mySignall);  

244     free(myFilterl);  

245     free(myConv1);  

246     free(myConv2);  

247     free(myConv3);  

248     free(myConv4);  

249  

250     // Free vectors on GPU  

251     cudaFree(dev_mySignal2);  

252     cudaFree(dev_myFilter2);  

253     cudaFree(dev_myConv2);  

254     cudaFree(dev_mySignal2);  

255     cudaFree(dev_myFilter2);  

256     cudaFree(dev_myConv2);  

257     cudaFree(dev_mySignal3);  

258     cudaFree(dev_myFilter3);  

259     cudaFree(dev_myConv3);  

260     cudaFree(dev_mySignal4);  

261     cudaFree(dev_myFilter4);  

262     cudaFree(dev_myConv4);  

263  

264     return 0;  

265 }
```


Chapter 4

System Overview

4.1 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 4.1. A block diagram of the equalization and symbol detector is shown in Figure 4.1. 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 5.

Figure 4.1: A block diagram of the estimation process.

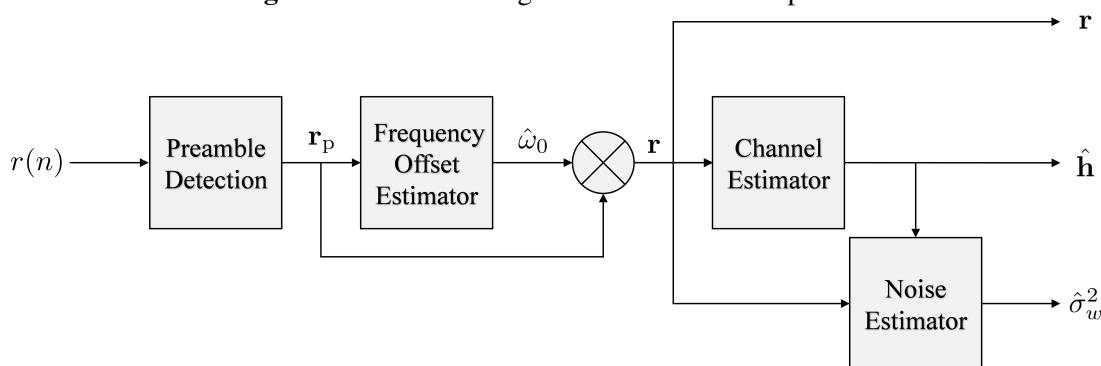


Figure 4.2: A block diagram of the equalization and symbol detector process.

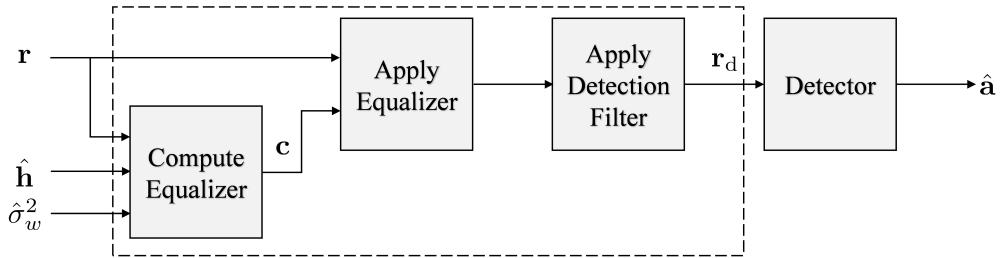
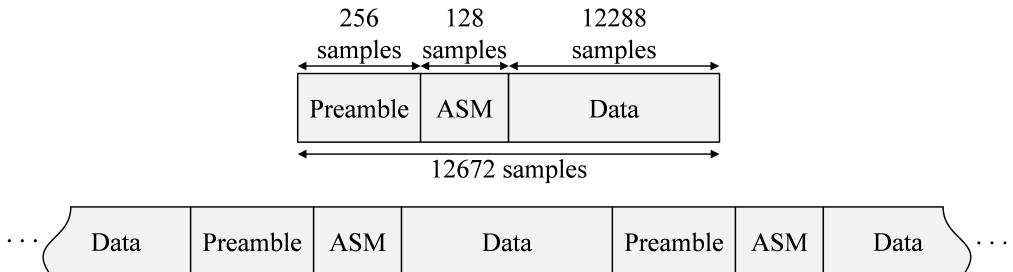


Figure 4.3: The iNET packet structure.



4.2 Preamble Detection

The received samples in this thesis has the iNET packet structure shown in Figure 4.3. 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

$$034776C72728950B0_{\text{hex}} \quad (4.1)$$

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 4.3. 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 [3]. Equations (4.2) through (4.5) have been optimized for GPUs and are implemented directly.

$$L(u) = \sum_{m=0}^7 [I^2(n, m) + Q^2(n, m)] \quad (4.2)$$

where the inner summations are

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

and

$$\begin{aligned} Q(n, m) \approx & \sum_{\ell \in \mathcal{L}_1} r_I(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_2} r_I(\ell + 32m + n) \\ & - \sum_{\ell \in \mathcal{L}_3} r_R(\ell + 32m + n) + \sum_{\ell \in \mathcal{L}_4} r_R(\ell + 32m + n) \\ & + 0.7071 \left[\sum_{\ell \in \mathcal{L}_5} r_I(\ell + 32m + n) - \sum_{\ell \in \mathcal{L}_6} r_I(\ell + 32m + n) \right. \\ & \left. - \sum_{\ell \in \mathcal{L}_7} r_R(\ell + 32m + n) + \sum_{\ell \in \mathcal{L}_8} r_R(\ell + 32m + n) \right] \end{aligned} \quad (4.4)$$

with

$$\begin{aligned}
\mathcal{L}_1 &= \{0, 8, 16, 24\} \\
\mathcal{L}_2 &= \{4, 20\} \\
\mathcal{L}_3 &= \{2, 10, 14, 22\} \\
\mathcal{L}_4 &= \{6, 18, 26, 30\} \\
\mathcal{L}_5 &= \{1, 7, 9, 15, 17, 23, 25, 31\} \\
\mathcal{L}_6 &= \{3, 5, 11, 12, 13, 19, 21, 27, 28, 29\} \\
\mathcal{L}_7 &= \{1, 3, 9, 11, 12, 13, 15, 21, 23\} \\
\mathcal{L}_8 &= \{5, 7, 17, 19, 25, 27, 28, 29, 31\}.
\end{aligned} \tag{4.5}$$

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

$$\mathbf{r}_p = \begin{bmatrix} r(k) \\ \vdots \\ r(k + L_{\text{pkt}} - 1) \end{bmatrix} = \begin{bmatrix} r_p(0) \\ \vdots \\ r_p(L_{\text{pkt}} - 1) \end{bmatrix} \tag{4.6}$$

4.3 Frequency Offset Compensation

The frequency offset estimator shown in Figure 4.1 is the estimator taken from [4, eq. (24)]. With the notation adjusted slightly, the frequency offset estimate is

$$\hat{\omega}_0 = \frac{1}{L_q} \arg \left\{ \sum_{n=i+2L_q}^{i+7L_q-1} r_p(n)r_p^*(n-L_q) \right\} \quad \text{for } i = 1, 2, 3, 4, 5. \tag{4.7}$$

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

$$r(n) = r_p(n)e^{-j\hat{\omega}_0}. \tag{4.8}$$

Equations (4.7) and (4.8) are easily implemented into GPUs.

4.4 Channel Estimation

The channel estimator is the ML estimator taken from [5, eq. 8].

$$\hat{\mathbf{h}} = \underbrace{(\mathbf{X}^\dagger \mathbf{X})^{-1} \mathbf{X}^\dagger}_{\mathbf{X}_{\text{lp}}^{-1}} \mathbf{r} \quad (4.9)$$

where \mathbf{X} is a convolution matrix formed from the ideal preamble and ASM samples and $\mathbf{X}_{\text{lp}}^{-1}$ is the left pseudo-inverse of \mathbf{X} . The ML channel estimator is the matrix operation

$$\hat{\mathbf{h}} = \mathbf{X}_{\text{lp}}^{-1} \mathbf{r}. \quad (4.10)$$

The matrix operation $\mathbf{X}_{\text{lp}}^{-1} \mathbf{r}$ maps simply and efficiently in GPUs using cuBLAS.

4.5 Noise Variance Estimation

The noise variance estimator is also taken from [5, eq. 9]

$$\hat{\sigma}_w^2 = \frac{1}{2\rho} |\mathbf{r} - \mathbf{X}\hat{\mathbf{h}}|^2 \quad (4.11)$$

where

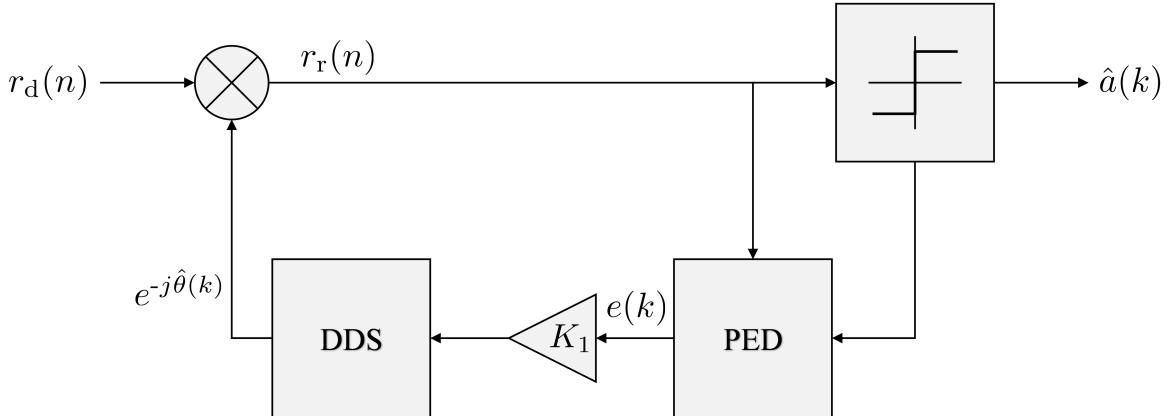
$$\rho = \text{Trace} \left\{ \mathbf{I} - \mathbf{X} (\mathbf{X}^\dagger \mathbf{X})^{-1} \mathbf{X}^\dagger \right\}. \quad (4.12)$$

Equation (4.11) is easily implemented into GPUs.

4.6 Symbol-by-Symbol Detector

The symbol by symbol detector block in Figure 4.2 is a Offset Quadrature Phase Shift Keying (OQPSK) detector. Using the simple OQPSK detector in place of the complex MLSE SOQPSK-TG detector leads to less than 1dB in bit error rate [6].

Figure 4.4: Offset Quadrature Phase Shift Keying symbol by symbol detector.



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

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

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.

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 5

Equalizer Equations

5.1 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.

5.2 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

$$\mathbf{R}\mathbf{c} = \hat{\mathbf{h}} \tag{5.1}$$

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} .

5.2.1 Zero-Forcing

The ZF equalizer is an FIR filter defined by the coefficients

$$c_{\text{ZF}}(-L_1) \quad \cdots \quad c_{\text{ZF}}(0) \quad \cdots \quad c_{\text{ZF}}(L_2). \quad (5.2)$$

The filter coefficients are the solution to the matrix vector equation [7, eq. (311)]

$$\mathbf{c}_{\text{ZF}} = (\mathbf{H}^\dagger \mathbf{H})^{-1} \mathbf{H}^\dagger \mathbf{u}_{n_0} \quad (5.3)$$

where

$$\mathbf{c}_{\text{ZF}} = \begin{bmatrix} c_{\text{ZF}}(-L_1) \\ \vdots \\ c_{\text{ZF}}(0) \\ \vdots \\ c_{\text{ZF}}(L_2) \end{bmatrix}, \quad (5.4)$$

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \left. \right\} n_0 - 1 \text{ zeros}, \quad (5.5)$$

$$\quad \quad \quad \left. \right\} N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros}$$

and

$$\mathbf{H} = \begin{bmatrix} \hat{h}(-N_1) & & & \\ \hat{h}(-N_1 + 1) & \hat{h}(-N_1) & & \\ \vdots & \vdots & \ddots & \\ \hat{h}(N_2) & \hat{h}(N_2 - 1) & \hat{h}(-N_1) & \\ & \hat{h}(N_2) & \hat{h}(-N_1 + 1) & \\ & & \vdots & \\ & & & \hat{h}(N_2) \end{bmatrix}. \quad (5.6)$$

Calculating Equation (5.3) 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 (5.3) results in

$$\begin{aligned} \mathbf{H}^\dagger \mathbf{H} \mathbf{c}_{\text{ZF}} &= \mathbf{H}^\dagger \mathbf{u}_{n_0} \\ \mathbf{R}_{\hat{h}} \mathbf{c}_{\text{ZF}} &= \hat{\mathbf{h}}_{n_0} \end{aligned} \quad (5.7)$$

where

$$\mathbf{R}_{\hat{h}} = \mathbf{H}^\dagger \mathbf{H} = \begin{bmatrix} r_{\hat{h}}(0) & r_{\hat{h}}^*(1) & \cdots & r_{\hat{h}}^*(L_{eq} - 1) \\ r_{\hat{h}}(1) & r_{\hat{h}}(0) & \cdots & r_{\hat{h}}^*(L_{eq} - 2) \\ \vdots & \vdots & \ddots & \\ r_{\hat{h}}(L_{eq} - 1) & r_{\hat{h}}(L_{eq} - 2) & \cdots & r_{\hat{h}}(0) \end{bmatrix} \quad (5.8)$$

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

$$r_{\hat{h}}(k) = \sum_{n=-N_1}^{N_2} \hat{h}(n) \hat{h}^*(n - k). \quad (5.9)$$

and

$$\hat{\mathbf{h}}_{n_0} = \mathbf{H}^\dagger \mathbf{u}_{n_0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix} \quad (5.10)$$

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

$$\mathbf{r}_{\hat{h}} = \begin{bmatrix} r_{\hat{h}}(0) \\ \vdots \\ r_{\hat{h}}(L_{ch}) \\ r_{\hat{h}}(L_{ch} + 1) \\ \vdots \\ r_{\hat{h}}(L_{eq} - 1) \end{bmatrix} = \begin{bmatrix} r_{\hat{h}}(0) \\ \vdots \\ r_{\hat{h}}(L_{ch}) \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad (5.11)$$

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

$$c_{\text{MMSE}}(-L_1) \quad \cdots \quad c_{\text{MMSE}}(0) \quad \cdots \quad c_{\text{MMSE}}(L_2). \quad (5.12)$$

The filter coefficients are the solution to the matrix vector equation [7, eq. (330) and (333)]

$$\mathbf{c}_{MMSE} = [\mathbf{G}\mathbf{G}^\dagger + 2\hat{\sigma}_w^2 \mathbf{I}_{L_1+L_2+1}]^{-1} \mathbf{g}^\dagger \quad (5.13)$$

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

$$\mathbf{G} = \begin{bmatrix} \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \\ & \ddots & \\ \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \\ & \ddots & \ddots \\ & & \hat{h}(N_2) & \cdots & \hat{h}(-N_1) \end{bmatrix} \quad (5.14)$$

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

$$\mathbf{g}^\dagger = \hat{\mathbf{h}}_{n0} = \begin{bmatrix} \hat{h}^*(L_1) \\ \vdots \\ \hat{h}^*(0) \\ \vdots \\ \hat{h}^*(-L_2) \end{bmatrix}. \quad (5.15)$$

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 (5.8) and defining

$$\mathbf{R}_{\hat{h}w} = \mathbf{R}_{\hat{h}} + 2\hat{\sigma}_w^2 \mathbf{I}_{L_1+L_2+1} = \begin{bmatrix} r_h(0) + 2\hat{\sigma}_w^2 & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) + 2\hat{\sigma}_w^2 & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) + 2\hat{\sigma}_w^2 \end{bmatrix}. \quad (5.16)$$

By placing Equation (5.16) and (5.15) into (5.13), solving for the MMSE equalizer coefficients \mathbf{c}_{MMSE} takes the form like the ZF equalizer coeffiencts in (5.7)

$$\mathbf{R}_{\hat{h}w} \mathbf{c}_{\text{MMSE}} = \hat{\mathbf{h}}_{n0}. \quad (5.17)$$

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 $\mathbf{R}_{\hat{h}}$.

5.2.2 The Constant Modulus Algorithm

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

$$c_{\text{CMA}(b)}(-L_1) \quad \dots \quad c_{\text{CMA}(b)}(0) \quad \dots \quad c_{\text{CMA}(b)}(L_2). \quad (5.18)$$

The filter coefficients are calculated by a steepest decent algorithm

$$\mathbf{c}_{\text{CMA}(b+1)} = \mathbf{c}_{\text{CMA}(b)} - \mu \nabla J \quad (5.19)$$

initialized by the MMSE equalizer coefficients

$$\mathbf{c}_{\text{CMA}(0)} = \mathbf{c}_{\text{MMSE}}. \quad (5.20)$$

The vector \mathbf{J} is the cost function and ∇J is the cost function gradient [7, eq. (352)]

$$\nabla J \approx \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \left[y(n)y^*(n) - 1 \right] y(n)\mathbf{r}^*(n). \quad (5.21)$$

where

$$\mathbf{r}(n) = \begin{bmatrix} r(n + L_1) \\ \vdots \\ r(n) \\ \vdots \\ r(n - L_2) \end{bmatrix}. \quad (5.22)$$

This means ∇J is defined by

$$\nabla J = \begin{bmatrix} \nabla J(-L_1) \\ \vdots \\ \nabla J(0) \\ \vdots \\ \nabla J(L_2) \end{bmatrix}. \quad (5.23)$$

A DSP engineer could implement Equation (5.21) and (5.19) directly but CMA can be massaged to map better to GPUs. To leverage the computational efficiency of convolution using the cuFFT library, Equation (5.21) is re-expressed as a convolution.

To begin messaging ∇J

$$z(n) = 2 \left[y(n)y^*(n) - 1 \right] y(n) \quad (5.24)$$

is defined to make the expression of ∇J to be

$$\nabla J = \frac{1}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} z(n) \mathbf{r}^*(n). \quad (5.25)$$

then writing the summation out in vector form

$$\nabla J = \frac{z(0)}{L_{pkt}} \begin{bmatrix} r^*(L_1) \\ \vdots \\ r^*(0) \\ \vdots \\ r^*(L_2) \end{bmatrix} + \frac{z(1)}{L_{pkt}} \begin{bmatrix} r^*(1+L_1) \\ \vdots \\ r^*(1) \\ \vdots \\ r^*(1-L_2) \end{bmatrix} + \dots + \frac{z(L_{pkt}-1)}{L_{pkt}} \begin{bmatrix} r^*(L_{pkt}-1+L_1) \\ \vdots \\ r^*(L_{pkt}-1) \\ \vdots \\ r^*(L_{pkt}-1-L_2) \end{bmatrix}. \quad (5.26)$$

The k th value of ∇J is

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m) r^*(m-k), \quad -L_1 \leq k \leq L_2. \quad (5.27)$$

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

$$\rho(n) = r^*(n). \quad (5.28)$$

Now

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)\rho(k-m). \quad (5.29)$$

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

$$\begin{aligned} b(n) &= \sum_{m=0}^{L_{pkt}-1} z(m)\rho(n-m) \\ &= \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-n) \end{aligned} \quad (5.30)$$

Comparing Equation (5.29) and (5.30) shows that

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2. \quad (5.31)$$

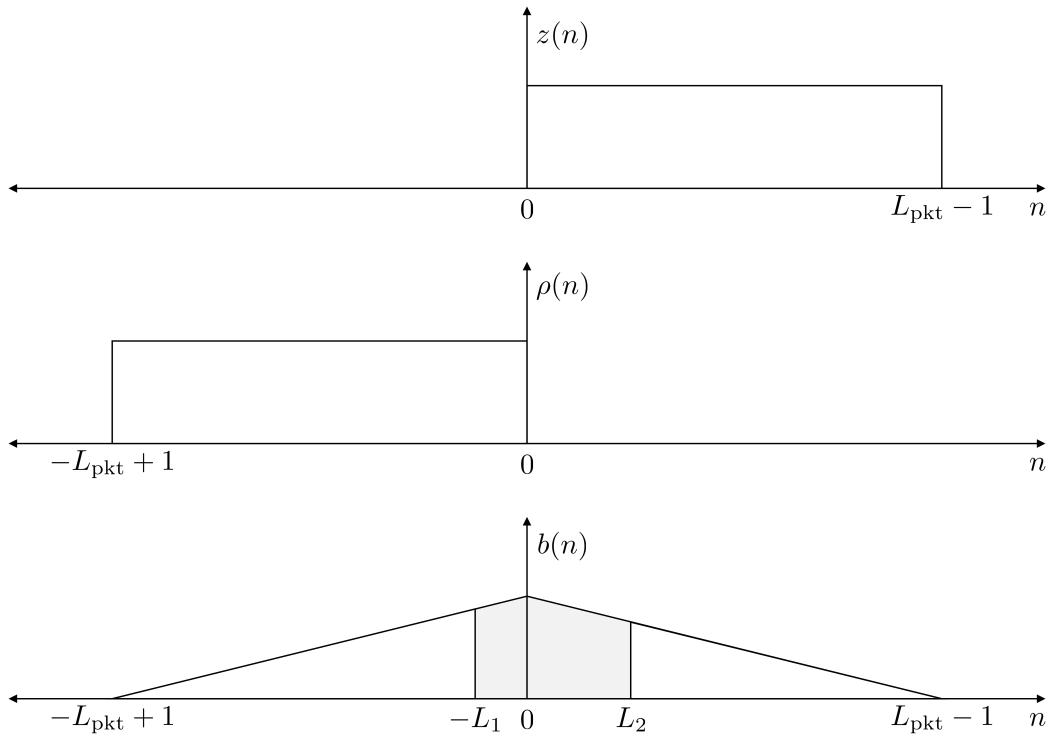
The values of interest are shown in Figure 5.1.

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

Listing 5.1: 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 5.1: 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

```

5.2.3 The Frequency Domain Equalizers

Williams and Saquib derived the Frequency Domain Equalizers (FDEs) for this application [8,].

The Frequency Domain Equalizer One

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 [8, eq. (11)]

$$C_{\text{FDE1}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1. \quad (5.32)$$

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.

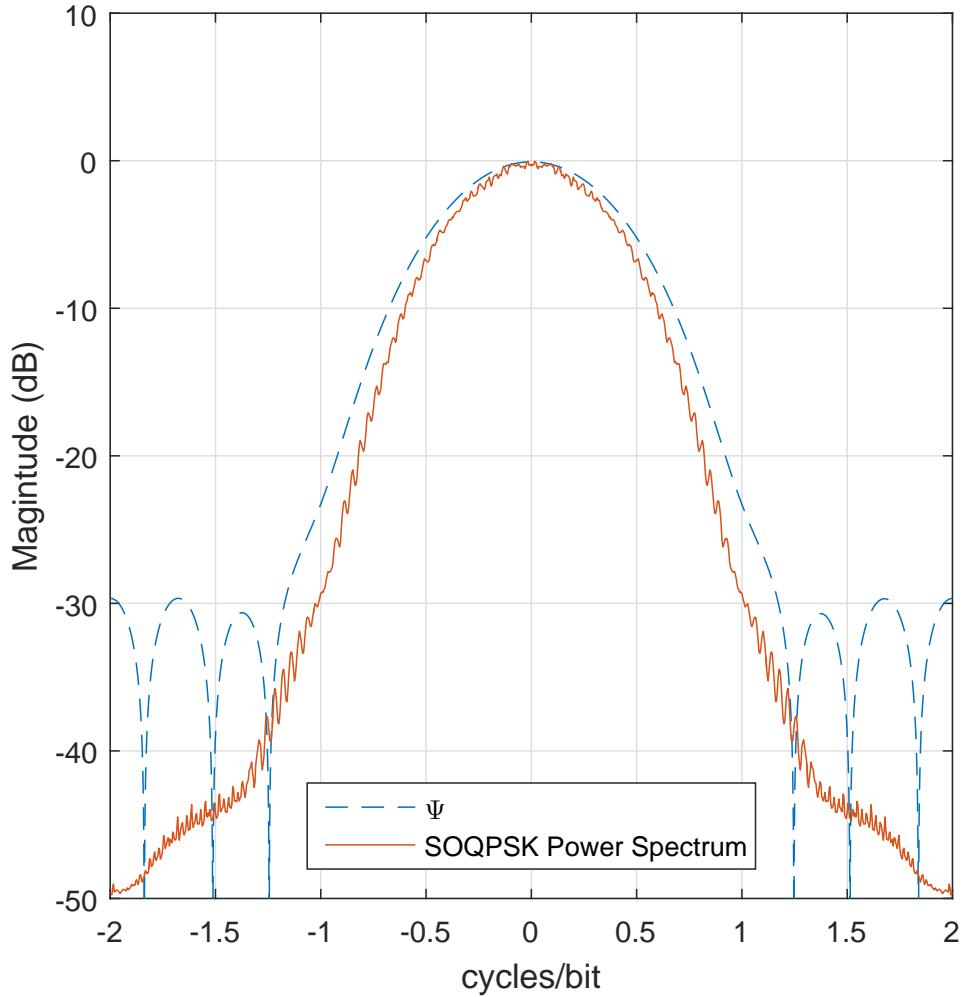
Equation (5.32) is straight forward to implement in GPUs. FDE1 is extremely fast and computationally efficient.

The Frequency Domain Equalizer One

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 [8, eq. (12)] The FDE2 equalizer is defined in Equation (12) as

$$C_{\text{FDE2}}(e^{j\omega_k}) = \frac{\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1 \quad (5.33)$$

Figure 5.2: I need help on this one!!!!



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 5.2.

Chapter 6

Equalizer GPU Implementation

In Chapter 5 the equalizer equations were conditioned for GPU implementation. This Chapter explains how the equalizers were implemented into GPUs. Block diagrams will be the main tool here because the code is too nasty to show a simple example.

Before we jump into the explanation of the equalizer implementation, we need to talk about a few things, namely Batched Processing in CUDA and convolution in GPUs.

6.1 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 6.1 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 [?]. Batched processing performs very well in GPUs because 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 6.2. If an operation is done on each packet, available

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

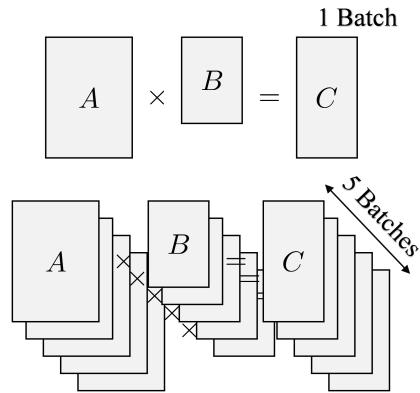
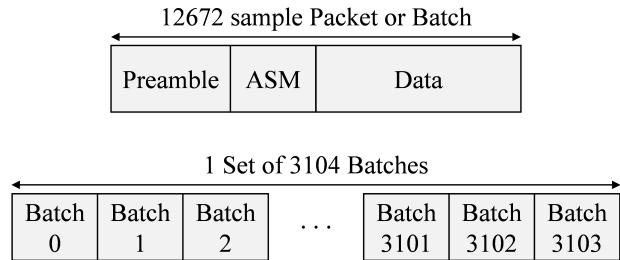


Figure 6.2: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.



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.

6.2 Batched Convolution

Chapter 3 delved into comparing a signal time domain and frequency domain convolution in CPUs and GPUs. Now 3104 equalizers needs to be convolved to the received signal packets.

$$\mathbf{r}_{d0} = \mathbf{r}_0 * \mathbf{c}_0 * \mathbf{h}_{NO} \quad (6.1)$$

$$\mathbf{r}_{d1} = \mathbf{r}_1 * \mathbf{c}_1 * \mathbf{h}_{NO} \quad (6.2)$$

$$\vdots \quad (6.3)$$

$$\mathbf{r}_{d3102} = \mathbf{r}_{3102} * \mathbf{c}_{3102} * \mathbf{h}_{NO} \quad (6.4)$$

$$\mathbf{r}_{d3103} = \mathbf{r}_{3103} * \mathbf{c}_{3103} * \mathbf{h}_{NO} \quad (6.5)$$

How does the problem change? Obviously doing batched convolution in a CPU will not be considered. Should we do convolution in the time domain or the frequency domain? If we do convolution in the time domain, should we use shared memory?

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.

6.3 Equalizer Implementations

Until now every all equations and block diagrams explain processing one packet of data. In the PAQ system, each batch of data has 3103 or 3104 packets in the $39321600 + 12671$ samples. Each equalizer in this chapter is processing one full batch of data. If the block diagram shows how to compute equalizer coefficients, assume that the block diagram is repeated 3103 or 3104 times in the GPU.

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 [6, Fig. 3].

6.4 Zero-Forcing and MMSE GPU Implementation

Computing the ZF and MMSE equalizer coefficients exactly the same as shown in Equations 5.7 and 5.17

$$\mathbf{R}_{\hat{h}} \mathbf{c}_{ZF} = \hat{\mathbf{h}}_{n_0} \quad (6.6)$$

$$\mathbf{R}_{\hat{h}w} \mathbf{c}_{MMSE} = \hat{\mathbf{h}}_{n_0}. \quad (6.7)$$

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.

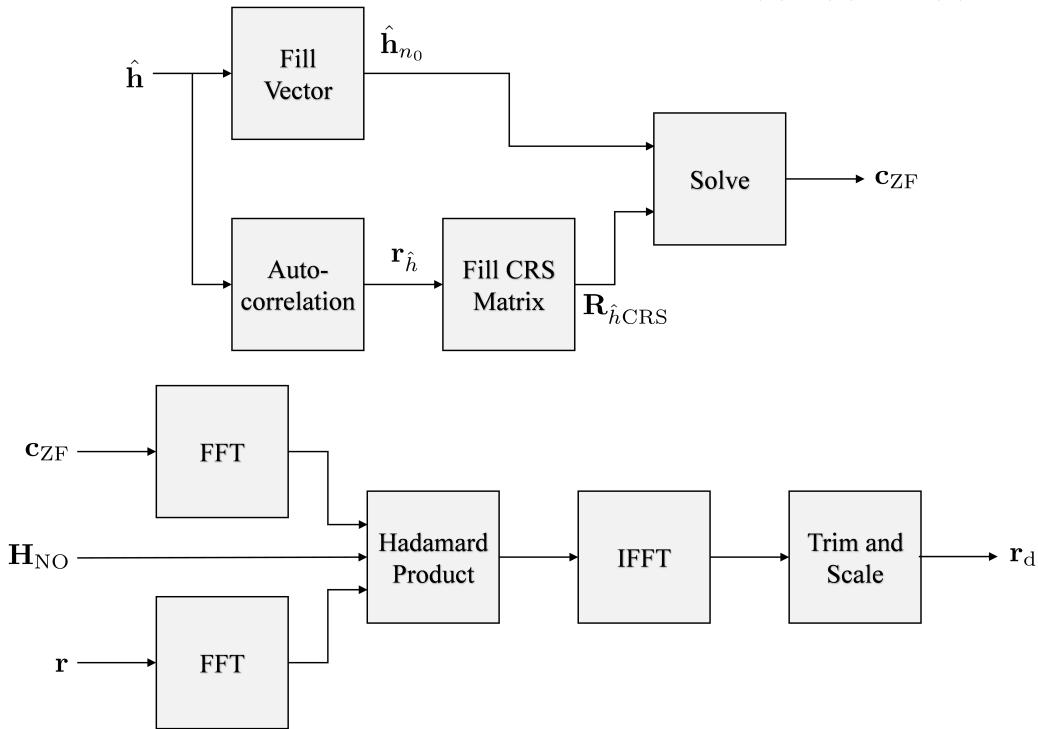
Zero-Forcing

Before solving Equation 6.6, $\mathbf{R}_{\hat{h}}$ 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 $\hat{\mathbf{h}}_{n_0}$.

Remember how $\mathbf{R}_{\hat{h}}$ is sparse? We now need to leverage the sparseness of $\mathbf{R}_{\hat{h}}$. With out leveraging the sparse properties of $\mathbf{R}_{\hat{h}}$, even the mighty Tesla K40c cannot produce \mathbf{c}_{ZF} in less than 1.907 seconds.

The sparseness of $\mathbf{R}_{\hat{h}}$ is to be leveraged by using a sparse solver function called “cusolver-SpCsrqrsBatched”. cusolverSpCsrqrsBatched is a batched complex solver that leverages the

Figure 6.3: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.



sparse properties of $R_{\hat{h}}$ by utilizing Compressed Row Storage (CRS) [9]. The large 186×186 matrix $R_{\hat{h}}$ is reduced to a 12544 element CSR matrix $R_{\hat{h}}^{CRS}$.

Once the filter coefficients are calculated, the filters c_{ZF} and H_{NO} are convolved with to r in the frequency domain.

MMSE

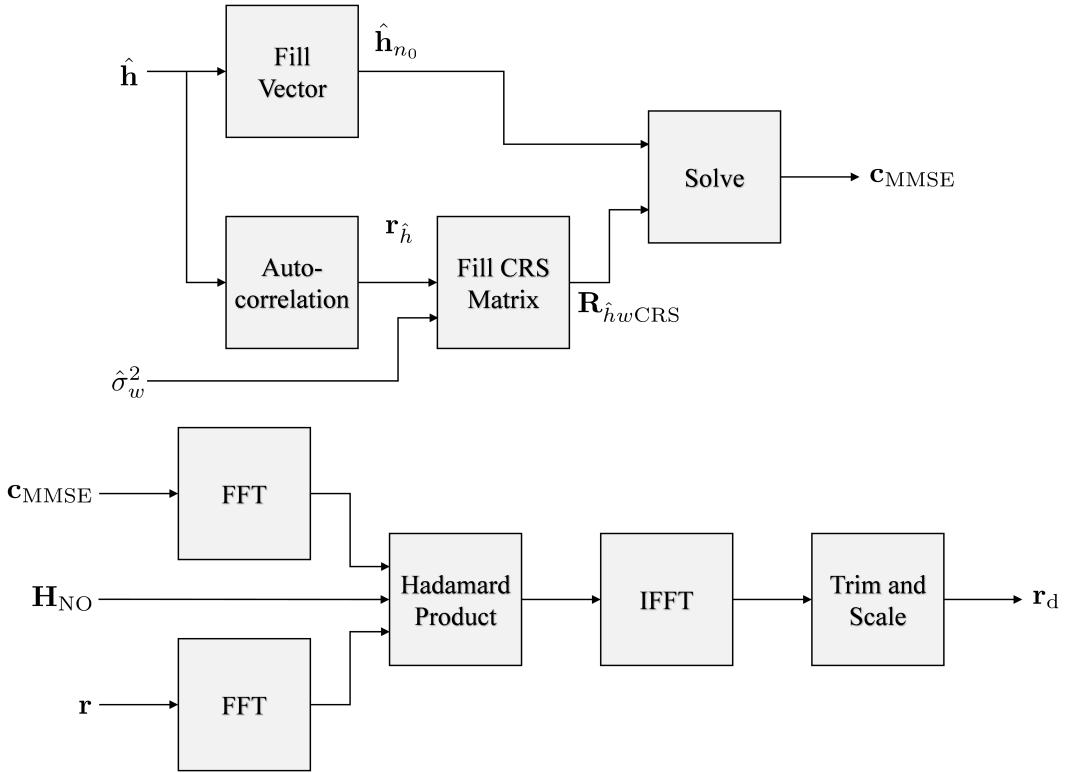
The MMSE equalizer coefficients are computed nearly identically to ZF accept when calculating $R_{\hat{h}w}^{CRS}$, $\hat{\sigma}_w^2$ is added to the main diagonal elements.

6.5 Constant Modulus Algorithm GPU Implementation

The Constant Modulus Algorithm is quite a bit more complicated than all other equalizers. The steps are this Apply the current filter c_{CMAb}

$$\mathbf{y} = \mathbf{r} * \mathbf{c}_{CMAb} \quad (6.8)$$

Figure 6.4: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.



Compute delJ

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2 \quad (6.9)$$

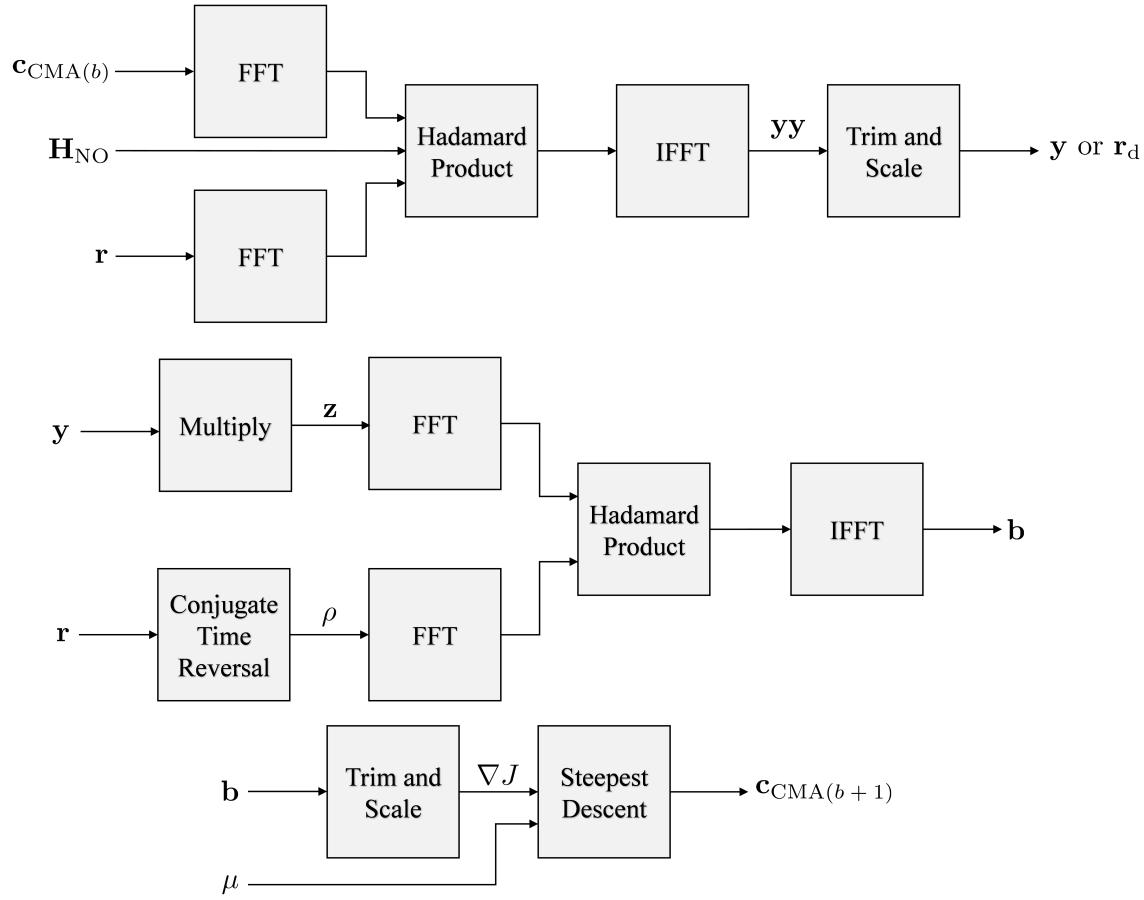
where

$$\begin{aligned} b(n) &= \sum_{m=0}^{L_{pkt}-1} z(m) \rho(n-m) \\ &= \sum_{m=0}^{L_{pkt}-1} z(m) r^*(m-n). \end{aligned} \quad (6.10)$$

Apply the steepest decent algorithm

$$\mathbf{c}_{\text{CMA}(b+1)} = \mathbf{c}_{\text{CMA}(b)} - \mu \nabla J. \quad (6.11)$$

Figure 6.5: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.



Once the number of CMA iterations is reached, apply the final CMA equalizer and the numerically optimized detection filter

$$r_d = r * c_{CMAb} * h_{NO}. \quad (6.12)$$

6.6 Frequency Domain Equalizer One and Two GPU Implementation

The Frequency Domain Equalizers are by far the fastest and easiest to implement into GPUs. Really, the block diagram looks just like convolution accept that hadamard product is replace with a different point to point multiply. We are already applying the filters in the frequency

Figure 6.6: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.

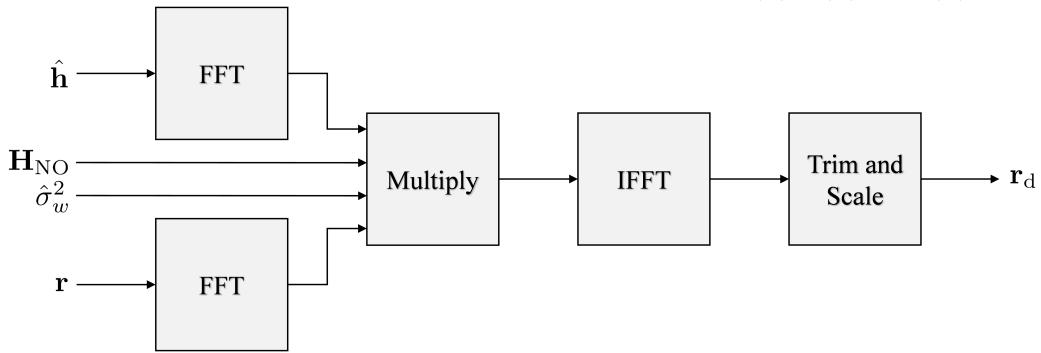
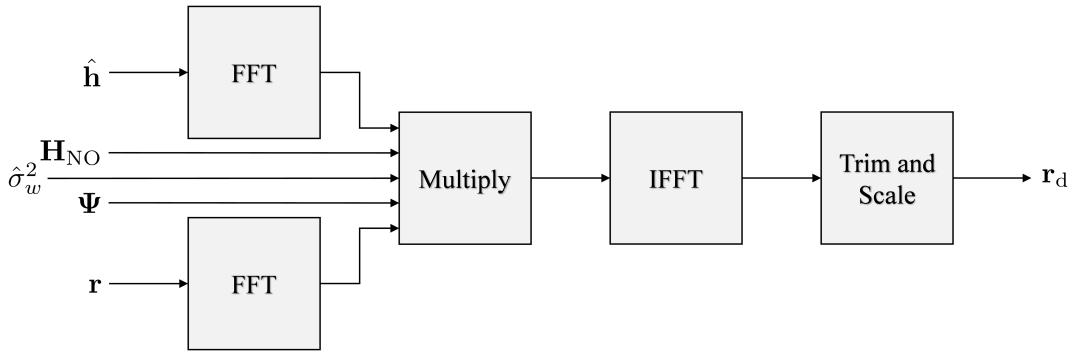


Figure 6.7: Diagram showing the relationships between $z(n)$, $\rho(n)$ and $b(n)$.



domain...so lets just wrap the equalizer calculation and detection filter application together by

$$R_d(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{1}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1 \quad (6.13)$$

or

$$R_d(e^{j\omega_k}) = \frac{R(e^{j\omega_k})\hat{H}^*(e^{j\omega_k})}{|\hat{H}(e^{j\omega_k})|^2 + \frac{\Psi(e^{j\omega_k})}{\hat{\sigma}_w^2}} \quad \text{where } \omega_k = \frac{2\pi}{L} \text{ for } k = 0, 1, \dots, L-1 \quad (6.14)$$

where $R(e^{j\omega_k})$ and $R_d(e^{j\omega_k})$ is the FFT r and r_d at ω_k .

Frequency Domain Equalizer One

Frequency Domain Equalizer Two

Chapter 7

Equalizer Performance

This is the Equalizer Performance

Chapter 8

Final Summary

this is the final summary

Bibliography

- [1] Wikipedia, “Graphics processing unit,” 2015. [Online]. Available: http://en.wikipedia.org/wiki/Graphics_processing_unit
- [2] NVIDIA, “Cuda toolkit documentation,” 2017. [Online]. Available: <http://docs.nvidia.com/cuda/>
- [3] 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.
- [4] 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.
- [5] 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.
- [6] E. Perrins, “FEC systems for aeronautical telemetry,” **IEEE Transactions on Aerospace and Electronic Systems**, vol. 49, no. 4, pp. 2340–2352, October 2013.
- [7] 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.
- [8] 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.
- [9] Wikipedia, “Sparse matrix,” 2017. [Online]. Available: https://en.wikipedia.org/wiki/Sparse_matrix