GPU Implementation of Data-Aided Equalizers

Jeffrey T. Ravert

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

Master of Science

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

Department of Electrical and Computer Engineering

Brigham Young University

April 2017

ABSTRACT

GPU Implementation of Data-Aided Equalizers

Jeffrey T. Ravert

Department of Electrical and Computer Engineering

Master of Science

Multipath is one of the dominant causes for link loss in aeronautical telemetry. Equalizers have been studied to combat multipath interference in aeronautical telemetry. Blind Constant Modulus Algorithm (CMA) equalizers are currently being used on SOQPSK-TG. The Preamble Assisted Equalization (PAQ) has been funded by the Air Force to study data-aided equalizers on SOQPSK-TG. PAQ compares side by side no equalization, data-aided zero forcing equalization, data-aided MMSE equalization, data-aided initialized CMA equalization, data-aided frequency domain equalization, and blind CMA equalization. An real time experimental test setup has been assembled including an RF receiver for data acquisition, FPGA for hardware interfacing and buffering, GPUs for signal processing, spectrum analyzer for viewing multipath events, and an 8 channel bit error rate tester to compare equalization performance. Lab tests were done with channel and noise emulators. Flight tests were conducted in March 2016 and June 2016 at Edwards Air Force Base to test the equalizers on live signals. The test setup achieved a 10Mbps throughput with a 6 second delay. Counter intuitive to the simulation results, the flight tests at Edwards AFB in March and June showed blind equalization is superior to data-aided equalization. Lab tests revealed some types of multipath caused timing loops in the RF receiver to produce garbage samples. Data-aided equalizers based on data-aided channel estimation leads to high bit error rates. A new experimental setup is been proposed, replacing the RF receiver with a RF data acquisition card. The data acquisition card will always provide good samples because the card has no timing loops, regardless of severe multipath.

Keywords: MISSING

ACKNOWLEDGMENTS

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

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This is the introduction

Chapter 2

Problem Statement

This is the Problem Statement

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

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

the ease of implementation vs HDL programming

the ease of setup

Chapter 3

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 explination 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

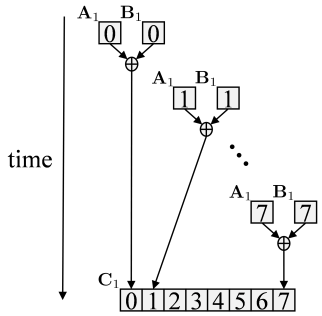


Figure 3.1: A block diagram of how a CPU sequentially performs vector addition.

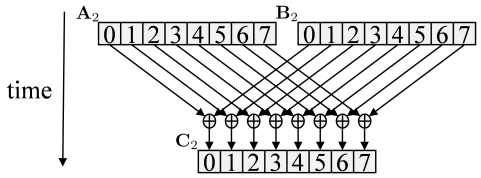


Figure 3.2: A block diagram of how a GPU performs vector addition in parallel.

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 to vectors together

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

where each vector is length 1024. Figure 3.1 shows how the CPU computes $\mathbf{C}_1$ by summing elements of $\mathbf{A}_1$ and $\mathbf{B}_1$ together *sequentially*. Figure 3.2 shows how the GPU computes $\mathbf{C}_2$ by summing elements of $\mathbf{A}_1$ and $\mathbf{B}_1$ 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 3.1: Comparison of CPU verse GPU code.

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

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

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

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

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


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

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

        // vector sum C1 = A1 + B1
        VecAddCPU(C1, A1, B1, numPoints);
        /*------------------------------------------------------------------
                                CPU End
        ------------------------------------------------------------------*/

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

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

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

```

```
67          // Set optimal number of threads per block
68          int numTreadsPerBlock = 32;
69
70          // Compute number of blocks for set number of threads
71          int numBlocks = numPoints/numTreadsPerBlock;
72
73          // If there are left over points, run an extra block
74          if(numPoints % numTreadsPerBlock > 0)
75                  numBlocks++;
76
77          // Run computation on device
78          //for(int i = 0; i < 100; i++)
79          VecAddGPU<<<numBlocks, numTreadsPerBlock>>>(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 numTreadsPerBlock$\times$numBlocks threads. Each thread has a unique index. CUDA calls this indes 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 = numTreadsPerBlock. Both threadIdx and blockIdx are three dimensional and have x, y and z components. In this thesis only the x dimension is used because GPU kernels operate only on vectors.

To replace a CPU for loop that runs 0 to $N-1$, a GPU kernel launches N threads are with T threads per thread block. The number of blocks need is $M = \frac{N}{T}$ or $M = \frac{N}{T} + 1$ if N is not an

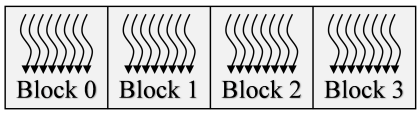


Figure 3.3: Block 0 32 threads launched in 4 thread blocks with 8 threads per block.

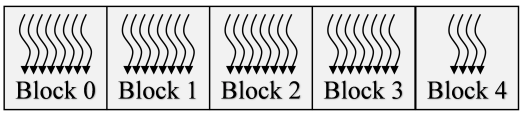


Figure 3.4: 36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.

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.

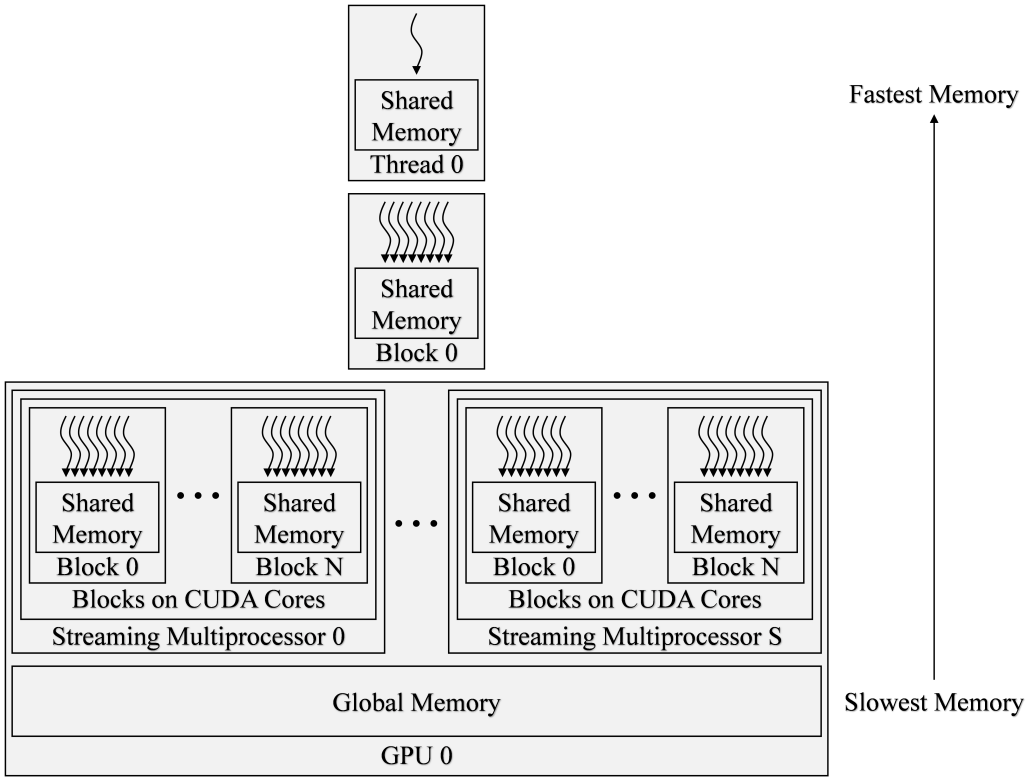
3.3 GPU memory

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.

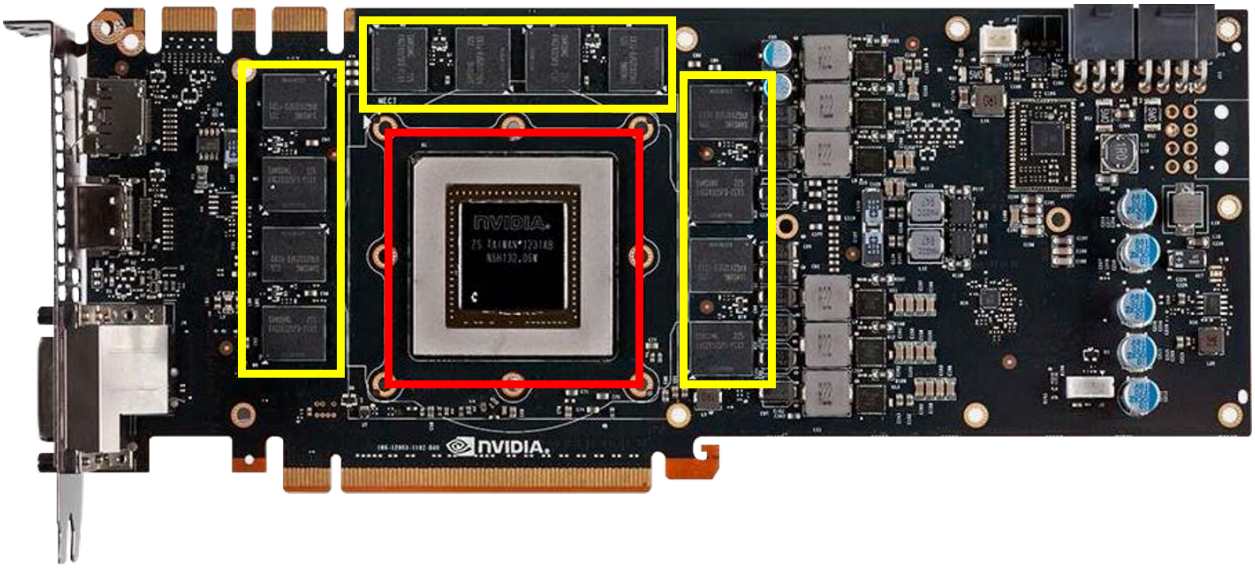


Figure 3.7: Example of an NVIDIA GPU card. The SRAM is shown to be boxed in yellow. The GPU chip is shown to be boxed in red.

3.4 Cuda Libraries

CUDA isn't just a programming language, it also has many libraries that are extreemly 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 any thing 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

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

where $x(m)$, $h(n-m)$ and $y(n)$ are complex. While Listing 3.1 is a simple and good example showing how 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 (3.2) 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 ConvGPUshared 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 verse frequency domain. Time domain convolution takes $\approx L_{\text{s}}L_{\text{h}}$ complex multiplies where L_{s} is the signal length and L_{h} is the filter length. Frequency domain convolution takes $\approx \frac{3N_{\text{FFT}}}{2}\log_2(N_{\text{FFT}})+ N_{\text{FFT}}$ complex multiplies where N_{FFT} is the next multiple of 2 above the convolution length $L_{\text{s}} + L_{\text{h}} - 1$.

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. If 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.

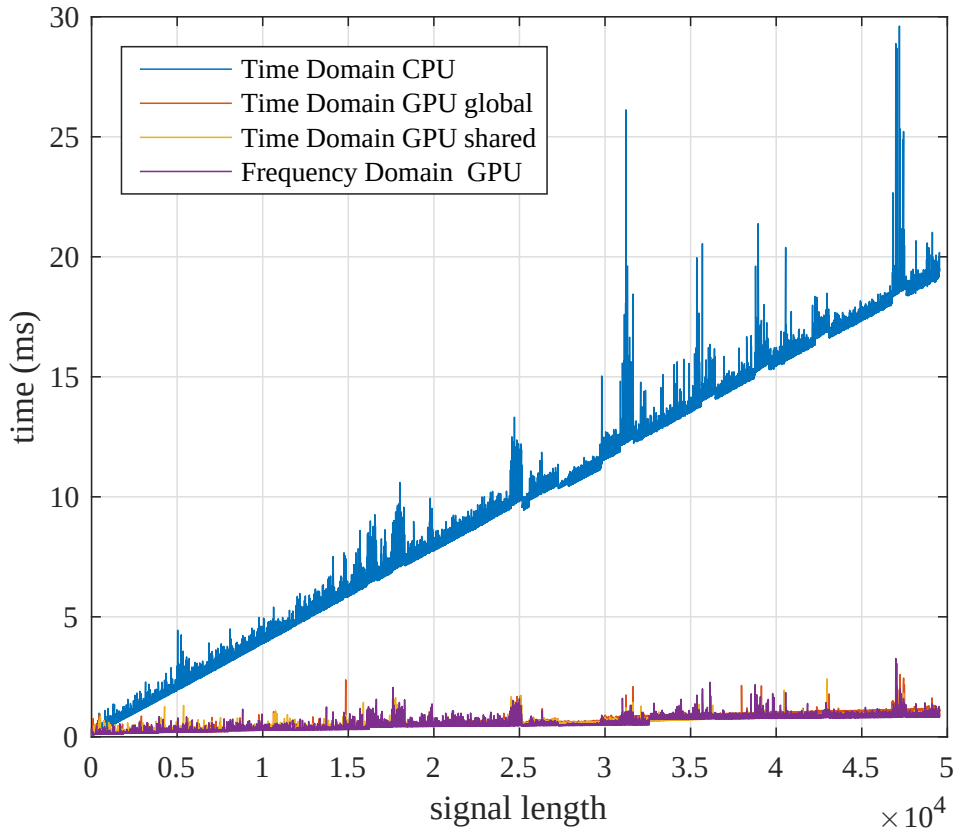


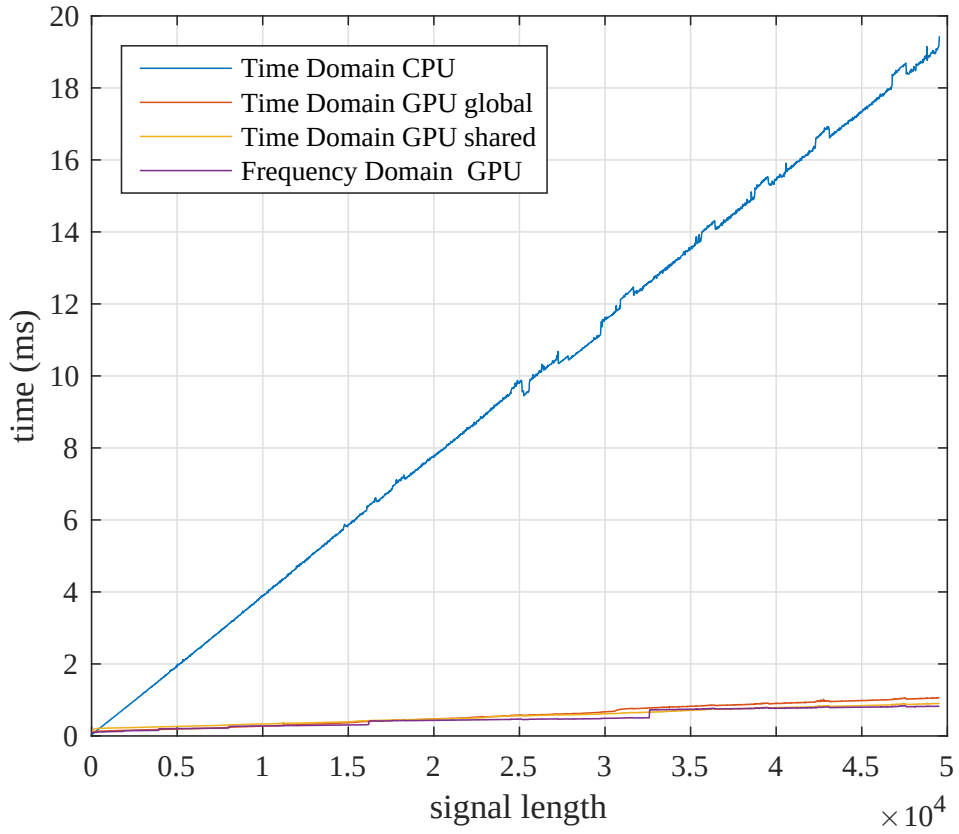
Table 3.3: Convolution computation times with signal length $2^{15} = 32768$ and filter length 186 on a Tesla K40c GPU.

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

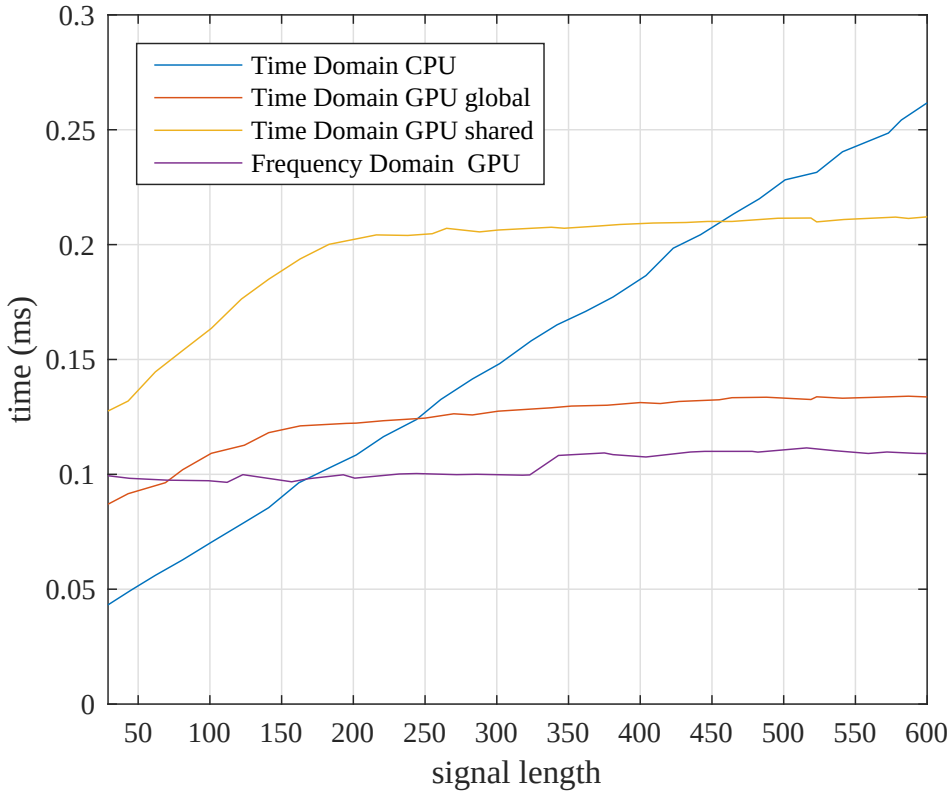
Figure 3.9: Comparison of complex convolution on CPU to GPU with varying signal lengths with lower bounding.



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.

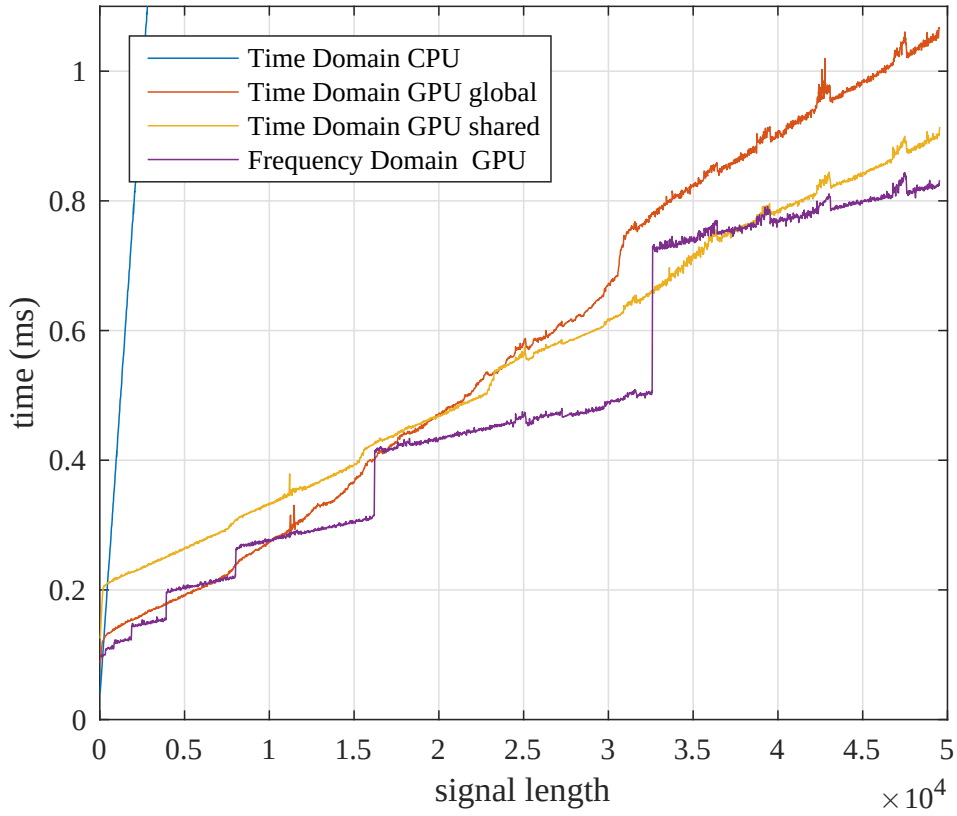


3.6 Thread Optimization

When first introduced to GPUs, a programmer might to tempted to launch as many threads per block 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.

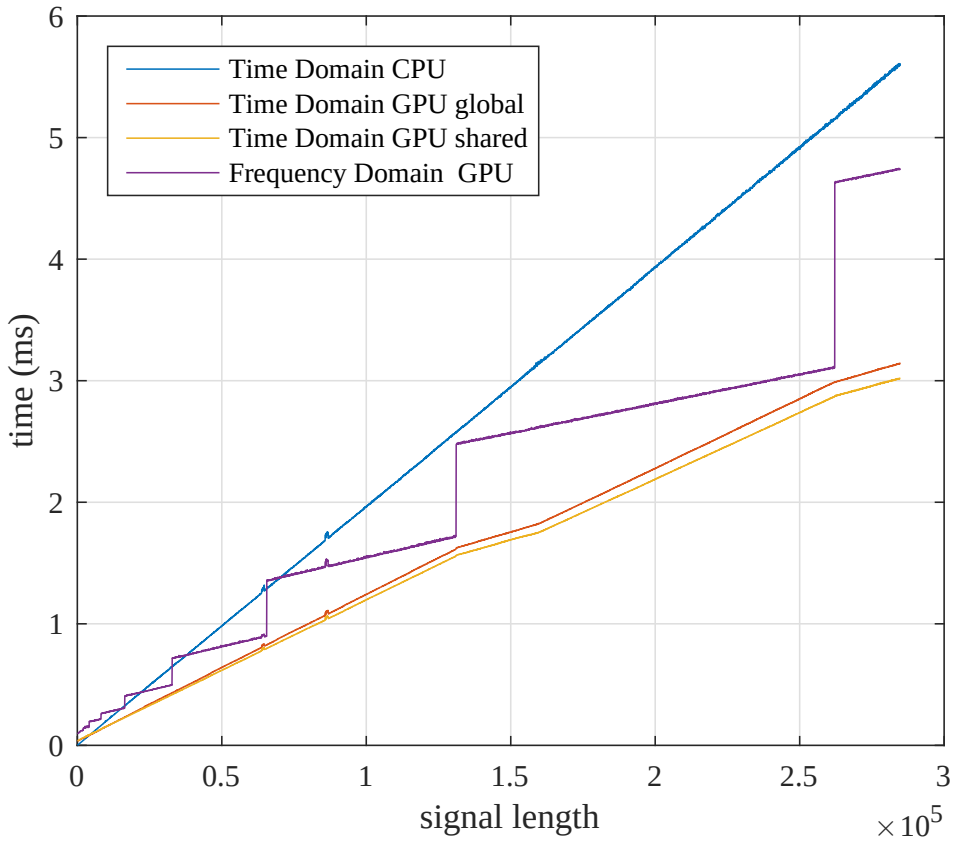


Knowing the perfect number of threads per block to launch is challenging to calculate. Luckily, there is a finite number of possible threads per block, 1 to 1024. A simple test program could time a GPU kernel while sweeping the number of threads per block from 1 to 1024. The number of threads per block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

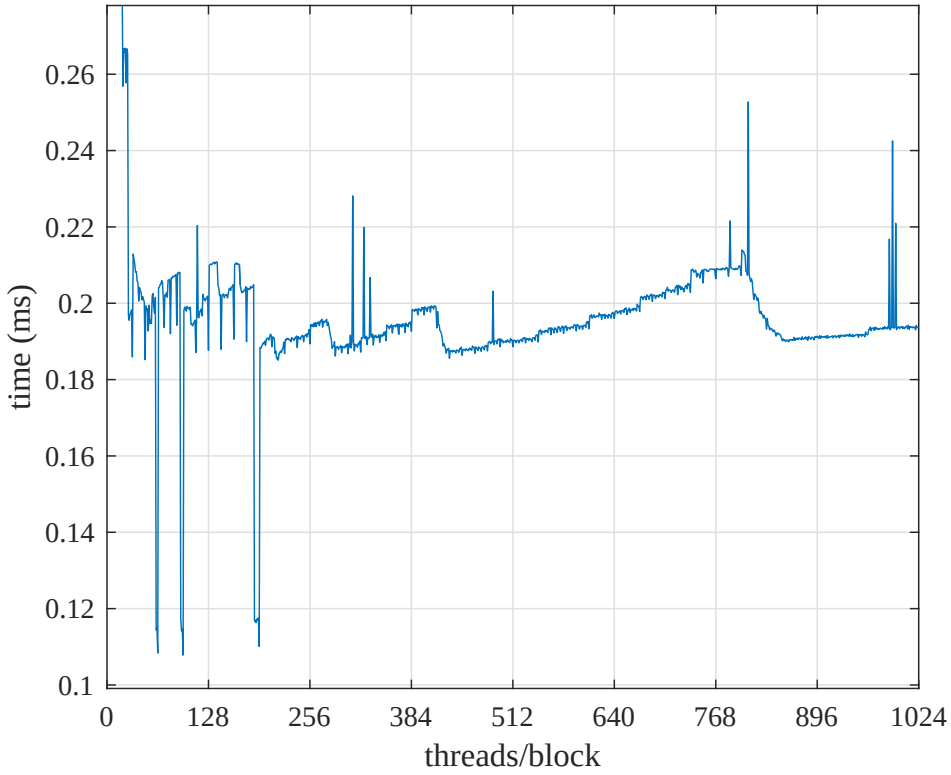


96 threads per block only transfer 96 filter coefficients to shared memory from global memory. Luckily, launching 192 threads per block is near optimal with an execution time of 0.1101ms. By simply adjusting the number of threads per block, ConvGPUshared can have a $2\times$ speed up.

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.

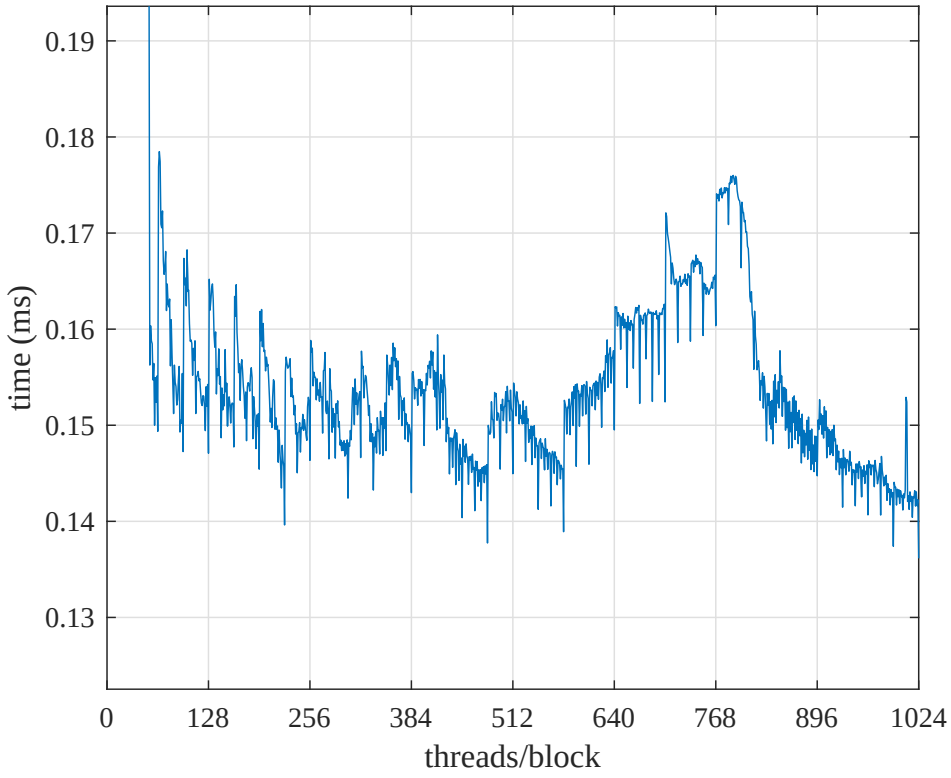
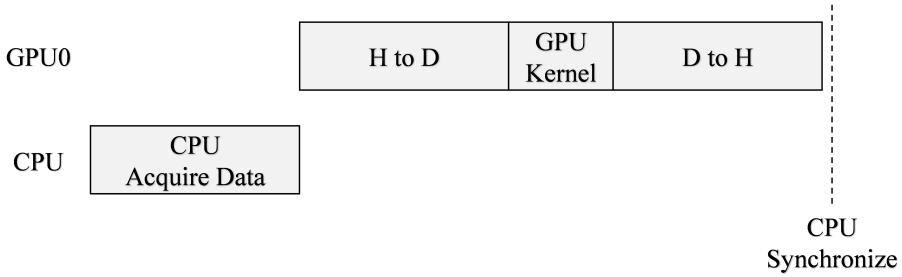


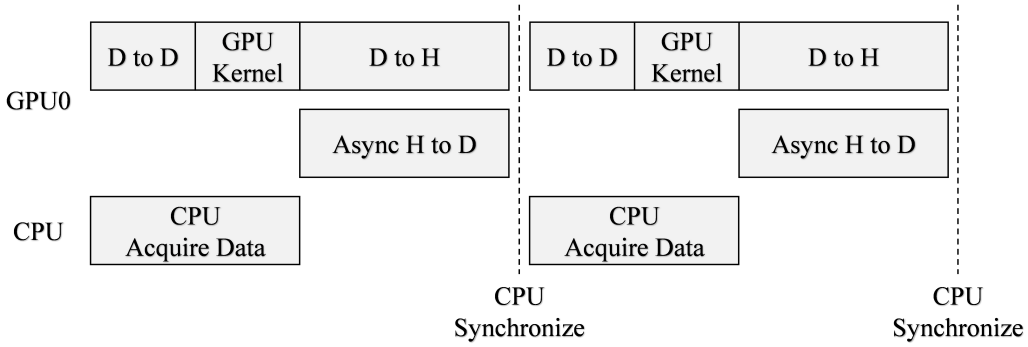
Figure 3.15: The typical approach of CPU and GPU operations. This block diagram shows a Profile of Listing 3.2.



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 Acuire 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  }
```

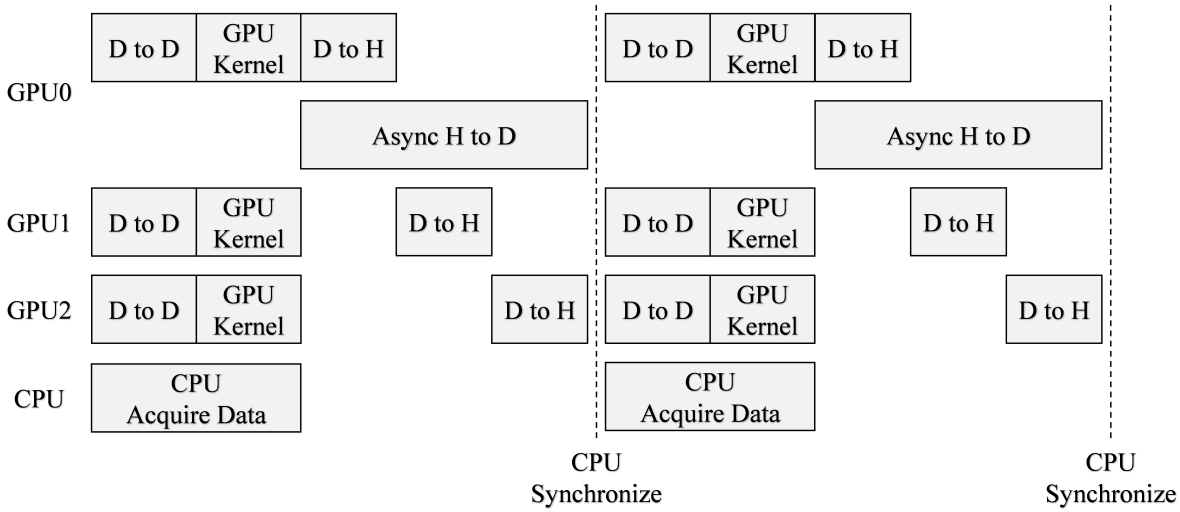
So the question is, "Can the throughput increase by using idle time on the GPU and CPU?" Yes, CPU and GPU operations can sacrifice latency for throughput by pipelineing. After the CPU gives instructions to the GPU, the CPU can do other operations like acquire data or perform algorithms better suited for a CPU than the GPU. Once the CPU has finished its operations, the CPU calls cudaDeviceSynchronize 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 Acuire 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 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 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.

```
#include <cufft.h>
using namespace std;
// Length of Filter (186 is a magic number)
const int LH = 186;

void ConvCPU(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
        for(int yIdx = 0; yIdx < Lx+Lh-1; yIdx++){
                cufftComplex temp;
                temp.x = 0;
                temp.y = 0;
                for(int hIdx = 0; hIdx < Lh; hIdx++){
                        int xAccessIdx = yIdx-hIdx;
                        if(xAccessIdx>=0 && xAccessIdx<Lx){
                                // temp += x[xAccessIdx]*h[hIdx];
                                // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
                                float A = x[xAccessIdx].x;
                                float B = x[xAccessIdx].y;
                                float C = h[hIdx].x;
                                float D = h[hIdx].y;
                                cufftComplex complexMult;
                                complexMult.x = A*C-B*D;
                                complexMult.y = A*D+B*C;

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

__global__ void ConvGPU(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
        int yIdx = blockIdx.x*blockDim.x + threadIdx.x;
        int lastThread = Lx+Lh-1;
        // Don't access elements out of bounds
        if(yIdx >= lastThread)
                return;
        cufftComplex temp;
        temp.x = 0;
        temp.y = 0;
        for(int hIdx = 0; hIdx < Lh; hIdx++){
                int xAccessIdx = yIdx-hIdx;
                if(xAccessIdx>=0 && xAccessIdx<Lx){
                        // temp += x[xAccessIdx]*h[hIdx];
                        // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
                        float A = x[xAccessIdx].x;
                        float B = x[xAccessIdx].y;
                        float C = h[hIdx].x;
                        float D = h[hIdx].y;
                        cufftComplex complexMult;
                        complexMult.x = A*C-B*D;
                        complexMult.y = A*D+B*C;

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

__global__ void ConvGPUshared(cufftComplex* y,cufftComplex* x,cufftComplex* h,int Lx,int Lh){
        int yIdx = blockIdx.x*blockDim.x + threadIdx.x;
        // Be sure to read in full h_shared before checking lastThread
        // First Lh threads in thread block read h from global memory into shared memory
        __shared__ cufftComplex h_shared[LH];
```

```
        if(threadIdx.x < LH)
                h_shared[threadIdx.x] = h[threadIdx.x];
        int lastThread = Lx+Lh-1;
        // Don't access elements out of bounds
        if(yIdx >= lastThread)
                return;
        // Thread barrier, ensures threads wait until h has been read into h_shared
        __syncthreads();
        cufftComplex temp;
        temp.x = 0;
        temp.y = 0;
        for(int hIdx = 0; hIdx < Lh; hIdx++){
                int xAccessIdx = yIdx-hIdx;
                if(xAccessIdx>=0 && xAccessIdx<Lx){
                        // temp += x[xAccessIdx]*h_shared[hIdx];

                        // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
                        float A = x[xAccessIdx].x;
                        float B = x[xAccessIdx].y;
                        float C = h_shared[hIdx].x;
                        float D = h_shared[hIdx].y;
                        cufftComplex complexMult;
                        complexMult.x = A*C-B*D;
                        complexMult.y = A*D+B*C;

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

__global__ void PointToPointMultiply(cufftComplex* vec0, cufftComplex* vec1, int lastThread){
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        // Don't access elements out of bounds
        if(i >= lastThread)
                return;
        // vec0[i] = vec0[i]*vec1[i]];
        // (A+jB)(C+jD) = (AC-BD) + j(AD+BC)
        float A = vec0[i].x;
        float B = vec0[i].y;
        float C = vec1[i].x;
        float D = vec1[i].y;
        cufftComplex complexMult;
        complexMult.x = A*C-B*D;
        complexMult.y = A*D+B*C;
        vec0[i] = complexMult;
}

__global__ void ScalarMultiply(cufftComplex* vec0, float scalar, int lastThread){
        int i = blockIdx.x*blockDim.x + threadIdx.x;
        // Don't access elements out of bounds
        if(i >= lastThread)
                return;
        cufftComplex scalarMult;
        scalarMult.x = vec0[i].x*scalar;
        scalarMult.y = vec0[i].y*scalar;
        vec0[i] = scalarMult;
}

int main(){
        int mySignalLength = pow(2,15);
        int myFilterLength = LH;
        int myConvLength   = mySignalLength + myFilterLength - 1;
        int Nfft        = pow(2, ceil(log(myConvLength)/log(2)));
        int numTreadsPerBlock;
        int numBlocks;

```

```
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,
                cudaMemcpyHostToDevice);
189         cudaMemcpy(dev_myFilter2, myFilter1, sizeof(cufftComplex)*myFilterLength,
                cudaMemcpyHostToDevice);
190         numTreadsPerBlock = 128;
191         numBlocks = myConvLength/numTreadsPerBlock;
192         if(myConvLength % numTreadsPerBlock > 0)
193                 numBlocks++;
194         ConvGPU<<<numBlocks, numTreadsPerBlock>>>(dev_myConv2, dev_mySignal2, dev_myFilter2,
                mySignalLength, myFilterLength);
195         cudaMemcpy(myConv2, dev_myConv2, myConvLength*sizeof(cufftComplex),
                cudaMemcpyDeviceToHost);
196 
197 
```

```
198         /*-------------------------------------------------------------------------------------
199                                 GPU Time Domain Convolution Using Shared Memory
200         -------------------------------------------------------------------------------------*/
201         cudaMemcpy(dev_mySignal3, mySignal1, sizeof(cufftComplex)*mySignalLength,
                cudaMemcpyHostToDevice);
202         cudaMemcpy(dev_myFilter3, myFilter1, sizeof(cufftComplex)*myFilterLength,
                cudaMemcpyHostToDevice);
203         numTreadsPerBlock = 192;
204         numBlocks = myConvLength/numTreadsPerBlock;
205         if(myConvLength % numTreadsPerBlock > 0)
206                 numBlocks++;
207         ConvGPUshared<<<numBlocks, numTreadsPerBlock>>>(dev_myConv3, dev_mySignal3, dev_myFilter3
                , mySignalLength, myFilterLength);
208         cudaMemcpy(myConv3, dev_myConv3, myConvLength*sizeof(cufftComplex),
                cudaMemcpyDeviceToHost);
209
210         /*-------------------------------------------------------------------------------------
211                                 GPU Frequency Domain Convolution using cuFFT
212         -------------------------------------------------------------------------------------*/
213         cudaMemset(dev_mySignal4, 0, Nfft*sizeof(cufftComplex));
214         cudaMemset(dev_myFilter4, 0, Nfft*sizeof(cufftComplex));
215         cudaMemcpy(dev_mySignal4, mySignal1, sizeof(cufftComplex)*mySignalLength,
                cudaMemcpyHostToDevice);
216         cudaMemcpy(dev_myFilter4, myFilter1, sizeof(cufftComplex)*myFilterLength,
                cudaMemcpyHostToDevice);
217         cufftExecC2C(plan, dev_mySignal4, dev_mySignal4, CUFFT_FORWARD);
218         cufftExecC2C(plan, dev_myFilter4, dev_myFilter4, CUFFT_FORWARD);
219         numTreadsPerBlock = 96;
220         numBlocks = Nfft/numTreadsPerBlock;
221         if(Nfft % numTreadsPerBlock > 0)
222                 numBlocks++;
223         PointToPointMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal4, dev_myFilter4, Nfft
                );
224         cufftExecC2C(plan, dev_mySignal4, dev_mySignal4, CUFFT_INVERSE);
225         numTreadsPerBlock = 128;
226         numBlocks = Nfft/numTreadsPerBlock;
227         if(Nfft % numTreadsPerBlock > 0)
228                 numBlocks++;
229         float scalar = 1.0/((float)Nfft);
230         ScalarMultiply<<<numBlocks, numTreadsPerBlock>>>(dev_mySignal4, scalar, Nfft);
231         cudaMemcpy(myConv4, dev_mySignal4, myConvLength*sizeof(cufftComplex),
                cudaMemcpyDeviceToHost);
232         cufftDestroy(plan);
233
234         // Free vectors on CPU
235         free(mySignal1);
236         free(myFilter1);
237         free(myConv1);
238         free(myConv2);
239         free(myConv3);
240         free(myConv4);
241
242         // Free vectors on GPU
243         cudaFree(dev_mySignal2);
244         cudaFree(dev_myFilter2);
245         cudaFree(dev_myConv2);
246         cudaFree(dev_mySignal2);
247         cudaFree(dev_myFilter2);
248         cudaFree(dev_myConv2);
249         cudaFree(dev_mySignal3);
250         cudaFree(dev_myFilter3);
251         cudaFree(dev_myConv3);
252         cudaFree(dev_mySignal4);
253         cudaFree(dev_myFilter4);
254         cudaFree(dev_myConv4);
255
256         return 0;
257 }
```

Chapter 4

System Overview

4.1 Overview

This chapter gives a high lever overview of the the algorithms implemented into the GPU. A block diagram is shown in Figure 4.1. The algorithms implemented in GPUs will briefly be explained. Chapter 5 explains the computation and application of the equalizers at a lower level. A simple block Diagram is shown in Figure 4.1.

This chapter will proceed as follows, section 4.2 will explain the algorithm used to find the preambles and packetize the received signal, section 4.3 will explain the frequency offset estimator and frequency offset compensation, section 4.4 will explain channel channel estimation, section 4.5 will explain noise variance, section 4.6 will explain the GPU implementation of the OQPSK detector. The explanation of the GPU implantation of the equalizers will be explained in much detail in Chapter 5.

4.2 Preamble Detection

The received samples in this project has the iNET packet structure shown in Figure 4.2. 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 a iNET packet L_{pkt} long or 12672 samples. The iNET preamble comprises eight repetitions of the 16-bit sequence CD98_{hex} and the ASM field

$$034776C72728950B0_{\text{hex}} \tag{4.1}$$

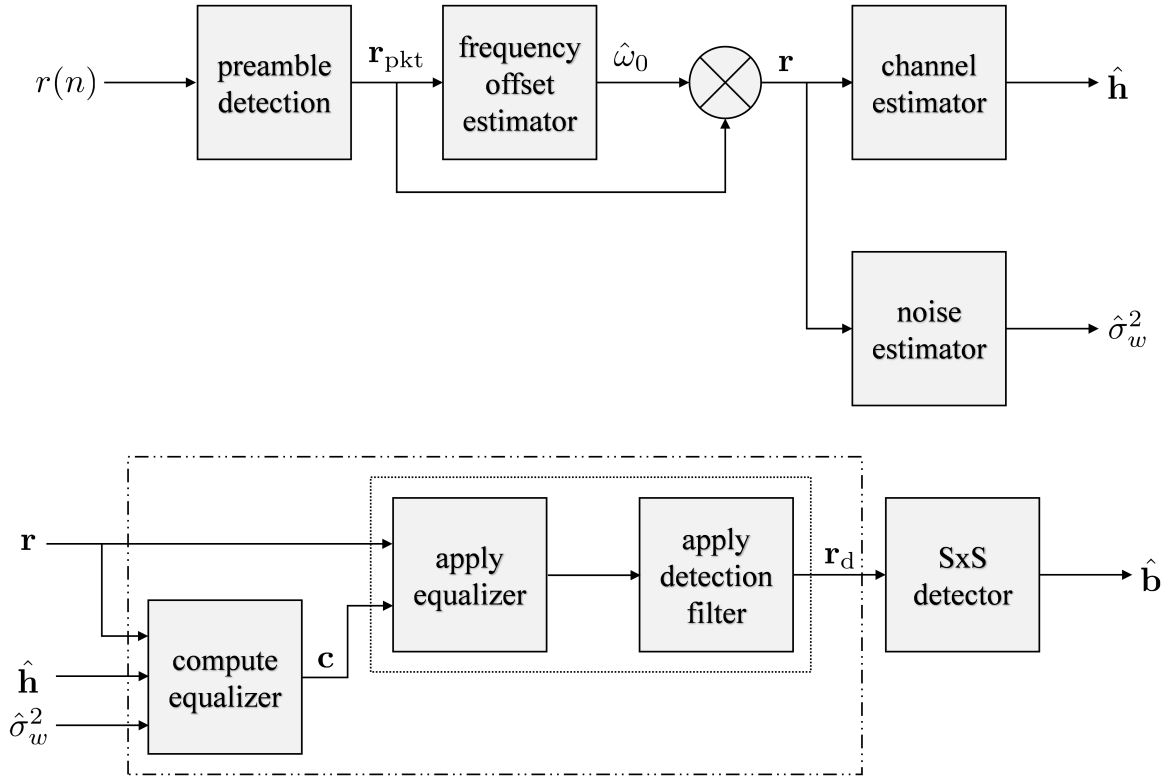


Figure 4.1: This a simple block diagram of what the GPU does.

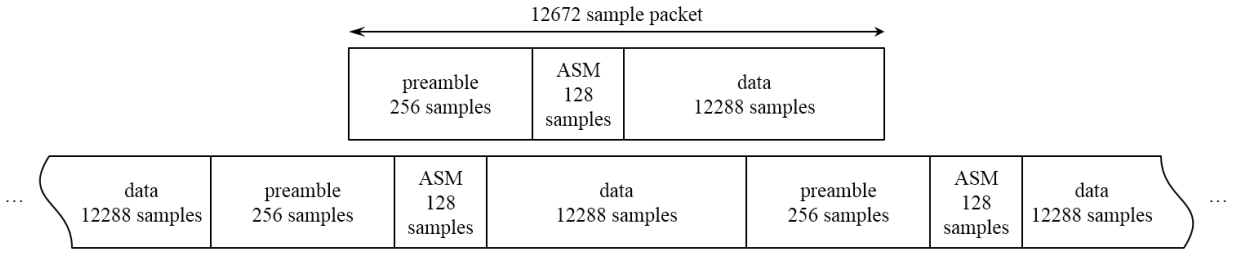


Figure 4.2: The iNET packet structure.

Each 16-bit sequence $\mathrm{CD98}_{\mathrm{hex}}$ sampled at two samples/bit are 32 or L_q 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 "packetize" the received samples into vectors with the packet structure shown in Figure 4.2. Each packet of received samples contains a L_{p} preamble samples, L_{ASM} ASM samples and L_{d} data

samples. The received signal is sampled at two samples per bit making $L_{\text{p}} = 256$, $L_{\text{ASM}} = 136$ and $L_{\text{d}} = 12288$. The full length of a packet is $L_{\text{p}} + L_{\text{ASM}} + L_{\text{d}} = 12672$.

Before the received samples can be packetized, the preambles are found using a preamble detector explained in [3]. The preamble detector output $L(u)$ is computed by

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

where the inner summations are

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

and

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

with

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

Figure 4.3 shows $2L_{\text{pkt}}$ samples of the preamble detector output $L(u)$. The start of a preamble is indicated by a local maximum of the preamble detector output. Using the index of the local maximums, the received samples are packetized. The vector $\mathbf{r}_{\text{pkt}}$ as shown in Figure 4.1 contains 12672 samples of data with the packet structure shown in Figure 4.2.

The preamble detection algoritm in Equations (4.2)-(4.5) and the local maximum search algorithms are easily implemented into GPUs. The GPU implementation of these algorithms wont be explained here.

4.3 Frequency Offset Compensation

The frequency offset estimator shown in Figure 4.1 is an algorithm taken from blah. With the notation adjusted slightly, the frequency offset estimate is

$$\hat{\omega}_0 = \frac{1}{L_q} \arg \left\{ \sum_{n=i+2L_q}^{i+7L_q-1} r(n) r^*(n - L_q) \right\} \tag{4.6}$$

where L_q is the length of where a frequency offset estimate is produced for every packet in $\mathbf{r}_{\text{pkt}}$.

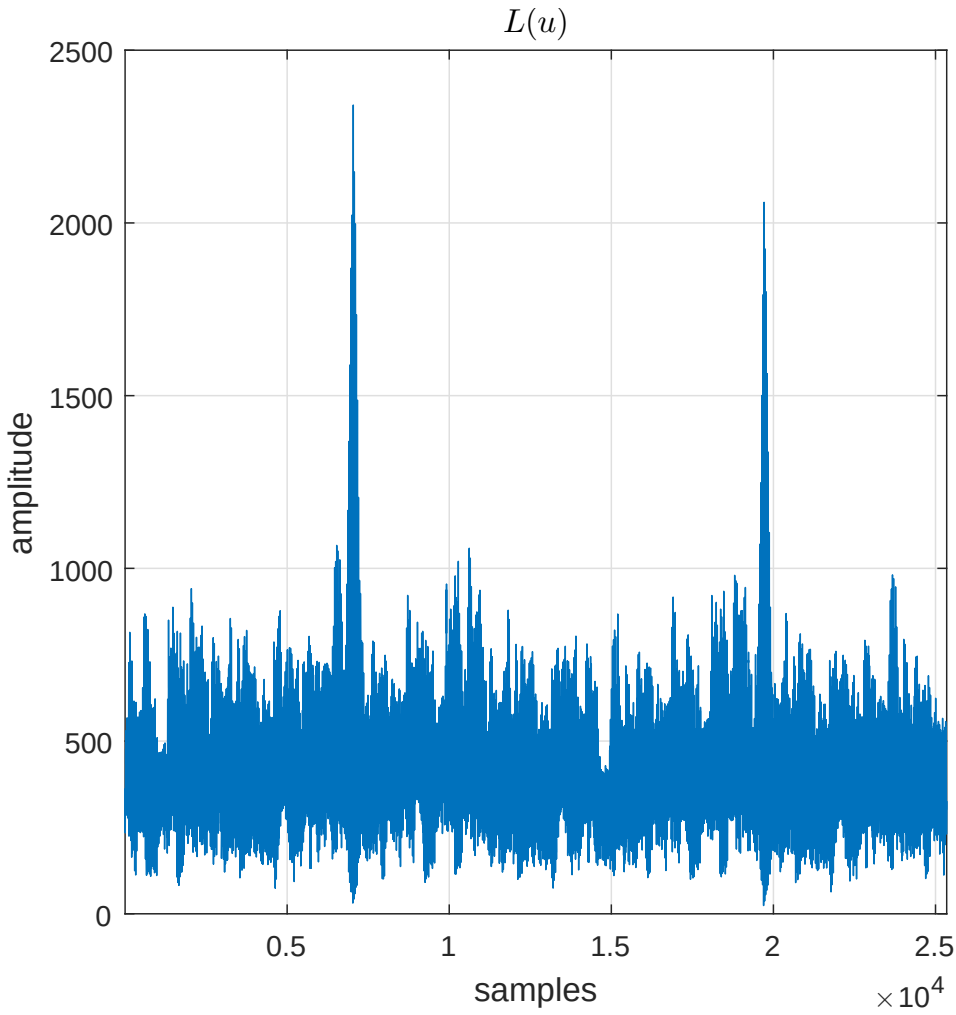


Figure 4.3: The output of the Preamble Detector $L(u)$.

The frequency offset is compensated for by derotating the packetized samples by $-\hat{\omega}_0$

$$r(n) = r_{\text{pkt}}(n)e^{-j\hat{\omega}_0} \tag{4.7}$$

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

4.4 Channel Estimation

The channel estimator is the ML estimator taken from blah.

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

where $\mathbf{X}$ is a convolution matrix formed from the ideal preamble and ASM samples. The matrix $\mathbf{P}_{\text{ix}}$ is

$$\mathbf{P}_{\text{ix}} = \left(\mathbf{X}^{\dagger}\mathbf{X}\right)^{-1}\mathbf{X}^{\dagger} \tag{4.9}$$

making the channel estimate simply

$$\hat{\mathbf{h}} = \mathbf{P}_{\text{ix}}\mathbf{r} \tag{4.10}$$

The matrix multiplication is easily implemented into GPUs.

4.5 Noise Variance Estimation

The noise variance estimator is the algorithm taken from blah. The algorithm is

$$\hat{\sigma}_w^2 = \frac{1}{2\rho}\left|\mathbf{r} - \mathbf{X}\hat{\mathbf{h}}\right|^2 \tag{4.11}$$

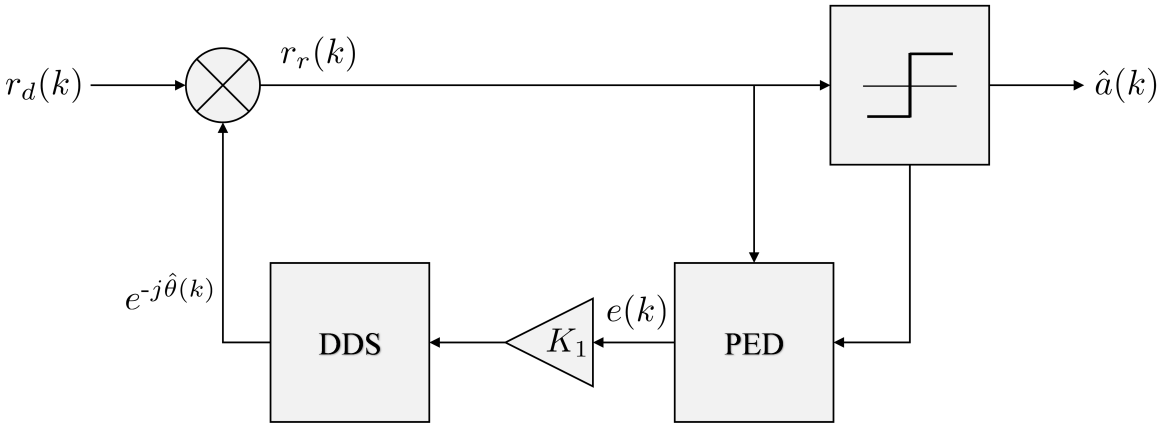
where ρ is the pre-computed constant

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

Equation (4.11) is easily implemented into GPUs.

4.6 SxS Detector

The Symbol by Symbol (SxS) detector block in Figure 4.1 is a Offset Quadtriture Phase Shift Keying (OQPSK) detector. Using the simple OQPSK detector in place of the complex MLSE SOQPSK-TG detector leads to less than 1 dB in bit error rate blah.



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

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

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

The Phase Lock Loop (PLL) in the SxS OQPSK detector cannot be parallelized to be implemented into GPUs because of the feedback loop. Feedback loops are inherently serial. Although the OQPSK detector 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 that performance and GPU implementation of 5 equalizers. While the performance and GPU implementation is interesting, this thesis makes no claim of theoretically expanding understanding of equalizers. The data-aided equalizers studied in this thesis are:

- Zero-Forcing (ZF)
- Minimum Mean Square Error (MMSE)
- Constant Modulus Algorithm (CMA)
- Frequency Domain Equalizer 1 (FDE1)
- Frequency Domain Equalizer 2 (FDE2)

The ZF and MMSE equalizers are very similar in formulation though from different sources. As you might tell from the names, FDE1 and FDE2 are 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 my solving linear equations

$$\mathbf{Rc} = \mathbf{h} \tag{5.1}$$

where $\mathbf{c}$ is vector of the desired equalizer coefficients and $\mathbf{R}$ is the auto-correlation matrix of the channel estimate $\mathbf{h}$. 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

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

The filter coefficients are the solution to the matrix vector equation [4, eq. (324)]

$$\mathbf{H}\mathbf{c}_{\text{ZF}} = \mathbf{u}_{n_0} \tag{5.3}$$

where

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

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} n_0 - 1 \text{ zeros} \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \end{matrix}, \tag{5.5}$$

and

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

Jeff explains how cuda solvers handle this equation.

The ZF equalizer was studied in the PAQ Phase 1 Final Report in equation 324

$$\mathbf{c}_{\text{ZF}} = (\mathbf{H}^{\dagger}\mathbf{H})^{-1}\mathbf{H}^{\dagger}\mathbf{u}_{n_0} \tag{5.7}$$

where $\mathbf{c}_{\text{ZF}}$ is a $L_{eq} \times 1$ vector of equalizer coefficients computed to invert the channel estimate $\mathbf{h}$ and $\mathbf{u}_{n_0}$ is the desired channel impulse response centered on $n0 = N_1 + L_1 + 1$

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} n_0 - 1 \text{ zeros} \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \end{matrix}. \tag{5.8}$$

The $L_{eq} + N_1 + N_2 \times L_{eq}$ convolution matrix $\mathbf{H}$ is built using the channel estimate $\mathbf{h}$

$$\mathbf{H} = \begin{bmatrix} h(-N_1) & & & \\ h(-N_1+1) & h(-N_1) & & \\ \vdots & \vdots & \ddots & \\ h(N_2) & h(N_2-1) & & h(-N_1) \\ & h(N_2) & & h(-N_1+1) \\ & & & \vdots \\ & & & h(N_2) \end{bmatrix}. \tag{5.9}$$

The computation of the coefficients in Equation (5.7) can be simplified in a couple of ways: First the matrix multiplication of $\mathbf{H}^\dagger$ and $\mathbf{H}$ is the autocorrelation matrix of the channel

$$\mathbf{R}_h = \mathbf{H}^\dagger\mathbf{H} = \begin{bmatrix} r_h(0) & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) \end{bmatrix} \tag{5.10}$$

where

$$r_h(k) = \sum_{n=-N_1}^{N_2} h(n)h^*(n-k). \tag{5.11}$$

Second the matrix vector multiplication of $\mathbf{H}^\dagger$ and $\mathbf{u}_{n_0}$ is simply the n_0th row of $\mathbf{H}^\dagger$ or the conjugated n_0th column of $\mathbf{H}$. A new vector $\mathbf{h}_{n_0}$ is defined by

$$\mathbf{h}_{n_0} = \mathbf{H}^\dagger\mathbf{u}_{n_0} = \begin{bmatrix} h(L_1) \\ \vdots \\ h(0) \\ \vdots \\ h(-L_2) \end{bmatrix}. \tag{5.12}$$

To simplify, Equations (5.10) and (5.12) are substituted into Equation (5.7) resulting in

$$\mathbf{c}_{\text{ZF}} = \mathbf{R}_h^{-1}\mathbf{h}_{n_0}. \tag{5.13}$$

Computing the inverse of $\mathbf{R}_h$ is computationally heavy because an inverse is an N^3 operation. To avoid an inverse, $\mathbf{R}_h$ is moved to the left side and $\mathbf{c}_{\text{ZF}}$ is found by solving a system of linear equations. Note that $r_h(k)$ only has support on $-L_{ch} \leq k \leq L_{ch}$ making $\mathbf{R}_h$ sparse or $\%63$ zeros. The sparseness of $\mathbf{R}_h$ is leveraged to reduce computation drastically. The Zero-Forcing Equalizer coefficients are computed by solving

$$\mathbf{R}_h\mathbf{c}_{\text{ZF}} = \mathbf{h}_{n_0}. \tag{5.14}$$

MMSE Equalizer

The MMSE equalizer has the same form as the Zero-Forcing equalizer. The MMSE equalizer was also studied in the PAQ Phase 1 Final Report in equation 330.

$$\mathbf{c}_{MMSE} = \left[\mathbf{G}\mathbf{G}^\dagger + \frac{\sigma_w^2}{\sigma_s^2}\mathbf{I}_{L_1+L_2+1}\right]\mathbf{g}^\dagger \tag{5.15}$$

where

$$\mathbf{G} = \begin{bmatrix} h(N_2) & \cdots & h(-N_1) & & & \\ & h(N_2) & \cdots & h(-N_1) & & \\ & & \ddots & & \ddots & \\ & & & h(N_2) & \cdots & h(-N_1) \end{bmatrix} \tag{5.16}$$

and

$$\mathbf{g} = \begin{bmatrix} h(L_1) \cdots h(-L_2) \end{bmatrix}. \tag{5.17}$$

The vector $\mathbf{g}^\dagger$ is also the same vector as $\mathbf{h}_{n_0}$ in Equation (5.8). The matrix multiplication $\mathbf{G}\mathbf{G}^\dagger$ is also the same autocorrelation matrix $\mathbf{R}_h$ as Equation (5.10). The fraction $\frac{1}{2\sigma_w^2}$ is substituted in for the fraction $\frac{\sigma_w^2}{\sigma_s^2}$ using Equation 333 Rice's report. MMSE only differs from Zero-Forcing by adding

the signal-to-noise ratio estimate down the diagonal of the autocorrelation matrix $\mathbf{R}_h$. Substituting in all these similarities in to Equation (5.15) results in

$$\Big[\mathbf{R}_h + \frac{1}{2\hat{\sigma_w^2}}\mathbf{I}_{L_1+L_2+1}\Big]\mathbf{c}_{\text{MMSE}} = \mathbf{h}_{n_0}. \tag{5.18}$$

To further simplify the notation, $\mathbf{R}_{hw}$ is substituted in for $\mathbf{R}_h + \frac{1}{2\hat{\sigma_w^2}}\mathbf{I}_{L_1+L_2+1}$ where

$$\mathbf{R}_{hw} = \mathbf{R}_h + \frac{1}{2\hat{\sigma_w^2}}\mathbf{I}_{L_1+L_2+1} = \begin{bmatrix} r_h(0) + \frac{1}{2\hat{\sigma_w^2}} & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) + \frac{1}{2\hat{\sigma_w^2}} & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) + \frac{1}{2\hat{\sigma_w^2}} \end{bmatrix}. \tag{5.19}$$

The MMSE equalizer coefficients are solved for in a similar fashion to the Zero-Forcing equalizer coefficients in Equation (5.14).

$$\mathbf{R}_{hw}\mathbf{c}_{\text{MMSE}} = \mathbf{h}_{n_0}. \tag{5.20}$$

5.2.2 The Iterative Equalizer

The Constant Modulus Algorithm

CMA uses a steepest decent algorithm.

$$\mathbf{c}_{b+1} = \mathbf{c}_b - \mu \nabla \mathbf{J} \tag{5.21}$$

The vector $\mathbf{J}$ is the cost function and ∇J is the cost function gradient defined in the PAQ report 352 by

$$\nabla J = \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \Big[y(n)y^*(n) - R_2 \Big] \, y(n)\mathbf{r}^*(n). \tag{5.22}$$

where

$$\mathbf{r}(n) = \begin{bmatrix} r(n+L_1) \\ \vdots \\ r(n) \\ \vdots \\ r(n-L_2) \end{bmatrix}. \tag{5.23}$$

This means ∇J is of the form

$$\nabla J = \begin{bmatrix} \nabla J(-L_1) \\ \vdots \\ \nabla J(0) \\ \vdots \\ \nabla J(L_2) \end{bmatrix}. \tag{5.24}$$

To leverage the computational efficiency of the Fast Fourier Transform (FFT), Equation (5.22) is re-expressed as a convolution.

To begin messaging ∇J

$$z(n) = 2 \Big[y(n)y^*(n) - R_2 \Big] \, y(n) \tag{5.25}$$

is defined to make the expression of ∇J to be

$$\nabla J = \frac{1}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} z(n)\mathbf{r}^*(n). \tag{5.26}$$

then writing the summation out in vector form

$$\nabla J = \frac{z(0)}{L_{pkt}} \begin{bmatrix} r^*(L_1) \\ \vdots \\ r^*(0) \\ \vdots \\ r^*(L_2) \end{bmatrix} + \frac{z(1)}{L_{pkt}} \begin{bmatrix} r^*(1+L_1) \\ \vdots \\ r^*(1) \\ \vdots \\ r^*(1-L_2) \end{bmatrix} + \cdots \frac{z(L_{pkt}-1)}{L_{pkt}} \begin{bmatrix} r^*(L_{pkt}-1+L_1) \\ \vdots \\ r^*(L_{pkt}-1) \\ \vdots \\ r^*(L_{pkt}-1-L_2) \end{bmatrix}. \tag{5.27}$$

The kth value of ∇J is

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-k), \quad -L_1 \leq k \leq L_2. \tag{5.28}$$

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

$$\rho(n) = r^*(n). \tag{5.29}$$

Now

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m)\rho(k-m). \tag{5.30}$$

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

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

$$= \sum_{m=0}^{L_{pkt}-1} z(m)r^*(m-n) \tag{5.31}$$

Comparing Equation (5.30) and (5.31) shows that

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2. \tag{5.32}$$

The values of interest are shown in Figure Foo!!!!(c)

This suggest the following algorithm for computing the gradient vector ∇J Matlab Code!!!

5.2.3 The Multiply Equalizers

The Frequency Domain Equalizer One

The Frequency Domain Equalizer One (FDE1) is the MMSE or wiener filter applied in the frequency domain. Ian E. Williams and M. Saquib derived FDE1 for this project in a paper called Linear Frequency Domain Equalization of SOQPSK-TG for Wideband Aeronautical Telemetry. The FDE1 equalizer is defined in Equation (11) as

$$C_{\text{FDE1}}(\omega) = \frac{\hat{H}^*(\omega)}{|\hat{H}(\omega)|^2 + \frac{1}{\hat{\sigma}^2}} \tag{5.33}$$

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

FDE1 needs no massaging because Equation (5.33) is easily implemented in the GPU and it is computationally efficient.

The Frequency Domain Equalizer One

The Frequency Domain Equalizer Two (FDE2) is the MMSE or wiener filter applied in the frequency domain. Ian E. Williams and M. Saquib derived FDE1 for this project in a paper called Linear Frequency Domain Equalization of SOQPSK-TG for Wideband Aeronautical Telemetry. The FDE2 equalizer is defined in Equation (12) as

$$C_{\text{FDE2}}(\omega) = \frac{\hat{H}^*(\omega)}{|\hat{H}(\omega)|^2 + \frac{\Psi(\omega)}{\hat{\sigma}^2}} \tag{5.34}$$

FDE2 almost identical to FDE1. The only difference is term $\Psi(\omega)$ in the denominator. The term $\Psi(\omega)$ is the average spectrum of SPQOSK-TG shown in Figure 5.1. FDE2 needs no massaging because Equation (5.34) is easily implemented in the GPU and is computationally efficient.

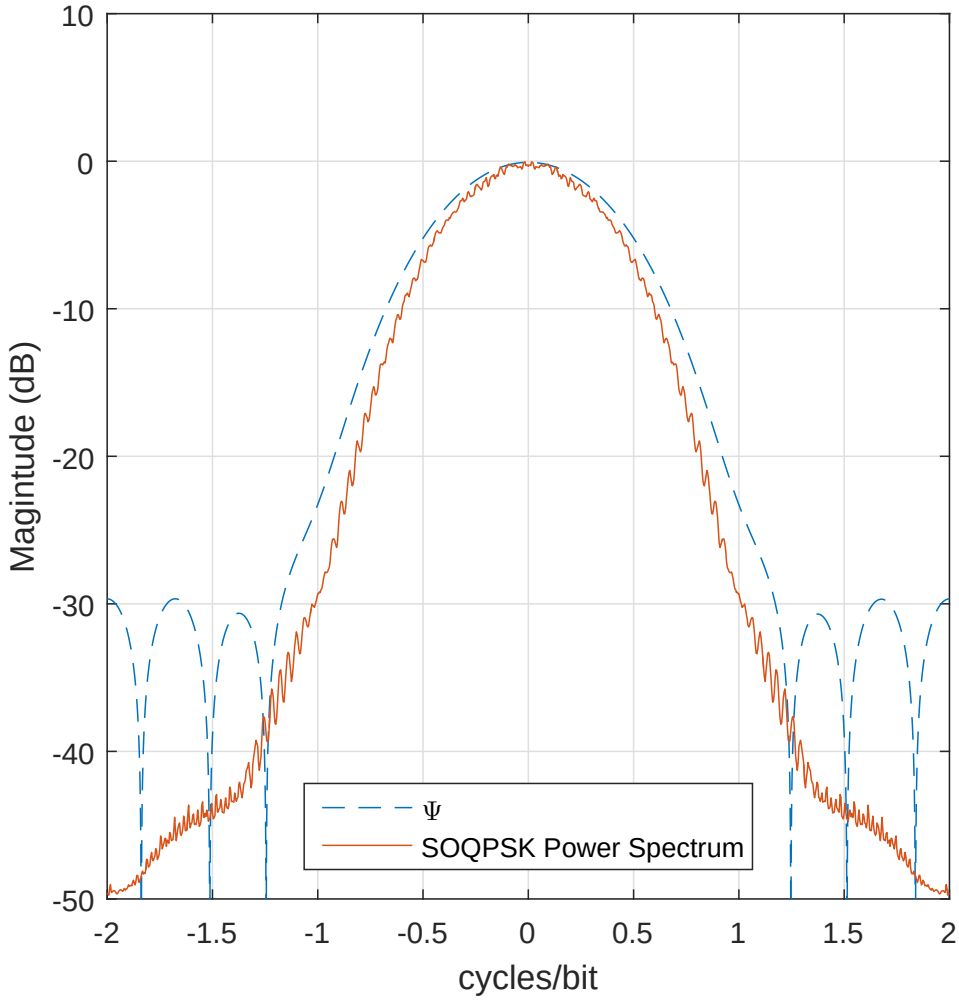


Figure 5.1: A block diagram illustrating organization of the algorithms in the GPU.

Chapter 6

Equalizer Performance

This is the Equalizer Performance

Chapter 7

Final Summary

this is the final summary

Bibliography

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

[2] NVIDIA, "Cuda toolkit documentation," 2017. [Online]. Available: http://docs.nvidia.com/cuda/ 5

[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. 31

[4] 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. 38