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

A Graphics Processing Unit (GPU) is a computational unit with a specialized, highly-parallel architecture well-suited to processing large blocks of data. The performance advantage derives from the GPUs ability to perform a large number of parallel computations on the large data block. Historically, the large block of data was graphics data and the computations were limited to those required for graphics operations. Since about 2006, there has been increasing interest in using genral purpose GPUs for more general processing tasks such as machine learning, oil exploration, scientific image processing, linear algebra, statistics, and 3D reconstruction [1]. This project leverages this trend and applies GPU processing to the estimation, computation, and filtering operations required for data-aided equalization.

Typically, the GPU architecture comprises a large memory block accessible by several (up to a few thousand) processing units simultaneously. For example, memory and processing units (called “CUDA cores”) for the two Nvidia GPUs used in this project are summarized in Table 2.1. Consequently, algorithms that are highly parallel are best suited for the GPU. In constrast, sequential algorithms (e.g., a phase lock loop!) are not well suited for the GPU.

Programming the Nvidia GPUs is written in the C++ computer language with API extensions known as CUDA (Compute Unified Device Architecture). The CUDA API is “a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements" [2]. CUDA allows the C++ program, running on the host CPU, to launch special functions—known as kernels— on the GPU while allowing some of the functionality to remain on the host CPU.

The choice to use GPUs, rather than FPGAs, for this project include the following:

1. The desire to test the performance of four equalizers operating in parallel on the received data is a good match to GPU structure.

2. Programming in C++ has advantages over VHDL designs in an FPGA. Development time is much shorter and debugging is quite a bit easier. Small modifications are much easier with C++ and GPUs than with VHDL in and FPGA.

3. Because the GPU supports floating point operations, the complications of tracking "digit growth" in fixed-point operations (on an FPGA) are eliminated.

GPUs are unlikely to be used for telemetry demodulators in the foreseeable future. However, because the point of the project is to assess the performance of competing algorithms, this fact is less important. Once identified, the "best" equalizer algorithm can be designed in VHDL and incorporated into the FPGA-based demodulators commonly found on the telemetry market.

The unique features of the GPU architecture require the designer to rethink how the signal processing is organized. DRAM memory limitations on the FPGA limit the number of samples per transfer $39{,}321{,}600$ complex-valued samples ($314{,}572{,}800$ Bytes). This data, corresponding to $3{,}103$ packets, is loaded into the GPU memory (cf. Table 2.1). Here starting indexes of the $3{,}103$ occurrences of the preamble are found. Subsequent signal processing for each packet is performed *in parallel*. In the end, $3{,}103 \times 6{,}144 = 19{,}064{,}832$ data bits *per equalizer* are produced at the end of the processing applied to each block. A conceptual block diagram of this organization is illustrated in Figure **??**. The preamble detector (or frame synchronizer), frequency offset estimator, channel estimator, and noise variance estimator are described in Section **??**. The equalizers are described in Section **??**.

Table 2.1: The computational resources available with three Nvidia GPUs used in this project (1x Tesla k40 2x Tesla K20).

Feature	Tesla k40	Tesla K20
Memory size (GDDR5)	12 GB	5 GB
CUDA cores	2880	2496

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 [3].

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

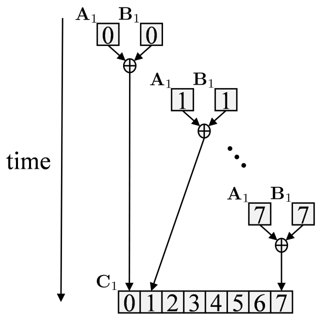


Figure 3.1: A block diagram of how a CPU sequentially performs vector addition.

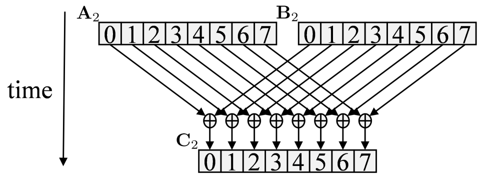


Figure 3.2: A block diagram of how a GPU performs vector addition in parallel.

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>
using namespace std;

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

__global__ void VecAddGPU(int* destination, int* source0, int* 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 = 1024;
        /*-------------------------------------------------------------------
                                    CPU Start
        -------------------------------------------------------------------*/
        // allocate memory on host
        int *A1;
        int *B1;
        int *C1;
        A1 = (int*) malloc (numPoints*sizeof(int));
        B1 = (int*) malloc (numPoints*sizeof(int));
        C1 = (int*) malloc (numPoints*sizeof(int));

        // 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
        int *C2;
        C2 = (int*) malloc (numPoints*sizeof(int));

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

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

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

        // Compute number of blocks for set number of threads
        int numBlocks = numPoints/numTreadsPerBlock;

        // If there are left over points, run an extra block
        if(numPoints % numTreadsPerBlock > 0)
                numBlocks++;

        // Run computation on device
        VecAddGPU<<<numBlocks, numTreadsPerBlock>>>(C2_gpu, A2_gpu, B2_gpu, numPoints);

        // Copy vector C2 from device to host
        cudaMemcpy(C2, C2_gpu, sizeof(int)*numPoints, cudaMemcpyDeviceToHost);
        /*----------------------------------------------------------------------
                                  GPU End
        ---------------------------------------------------------------------*/

        // Free vectors on CPU
        free(A1);
        free(B1);
        free(C1);
        free(C2);

        // Free vectors on GPU
        cudaFree(A2_gpu);
        cudaFree(B2_gpu);
        cudaFree(C2_gpu);
}
```

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 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.

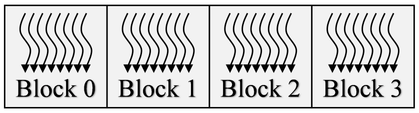


Figure 3.3: Block 0 32 threads launched in 4 thread blocks with 8 threads per block.

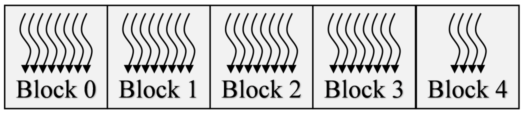


Figure 3.4: 36 threads launched in 5 thread blocks with 8 threads per block with 4 idle threads.

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, Figure 3.11 shows the form factor of the these GPUs. The red box in Figure 3.12 show the GPU 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?

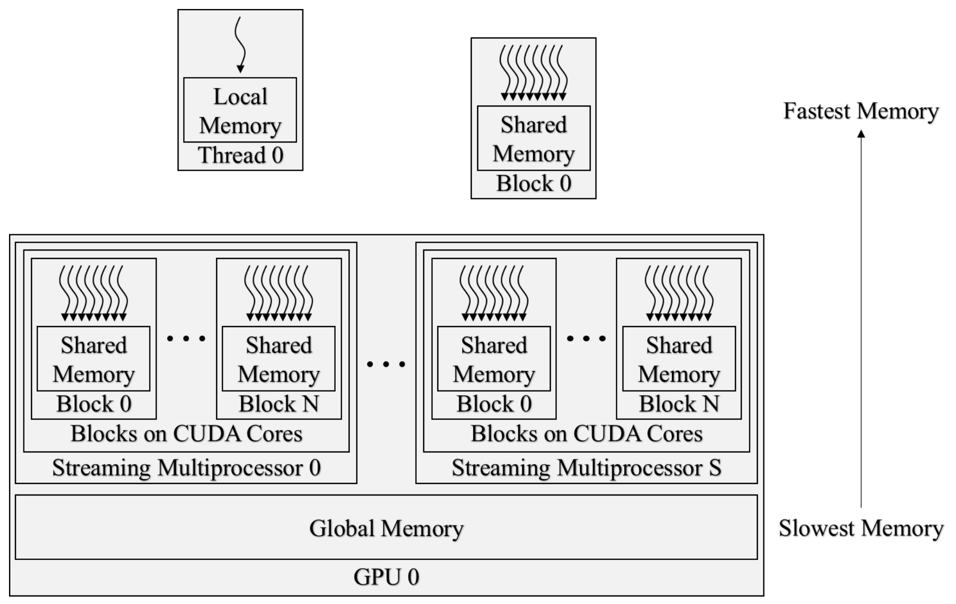


Figure 3.5: A block diagram where local, shared, and global memory is located. Each thread has private local memory. Each thread block has private shared memory. The GPU has global memory that all threads can access.

Figure 3.6: NVIDIA Tesla K40c and K20c.

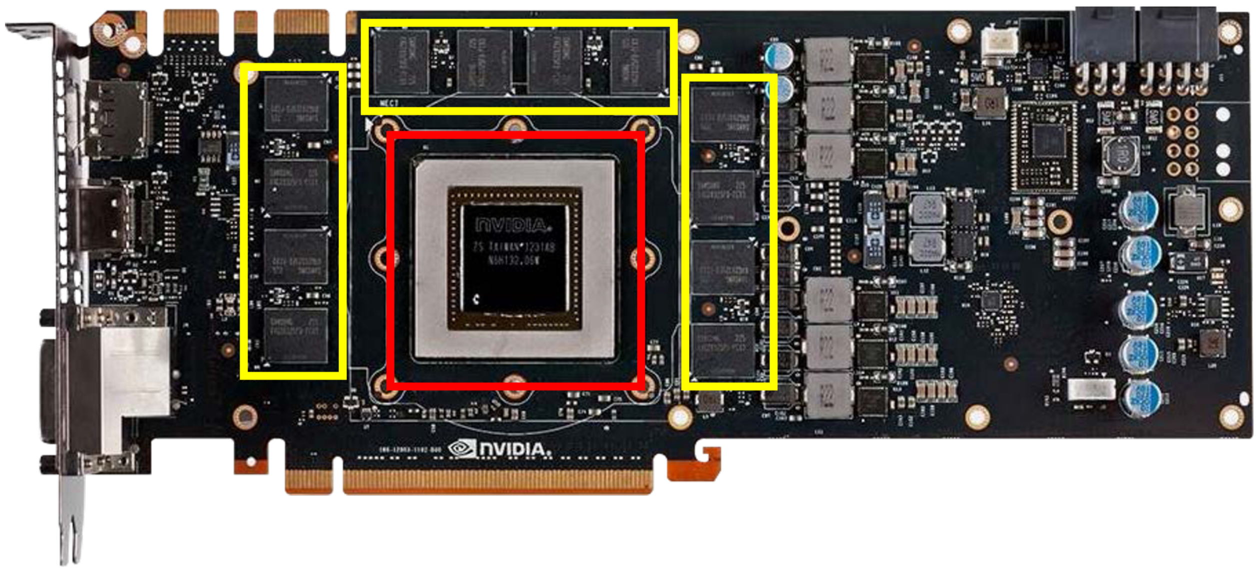


Figure 3.7: Example of an NVIDIA GPU card. The SRAM is shown to be boxed in yellow. The GPU chip is shown to be boxed in red.

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. The number of threads per block affect the amount of resources available to each thread. If a kernel is more computationally heavy, launching less threads per block may lead to faster execution time because threads have more resources available. A kernel can also be extremely computationally light, like VecAddGPU, making each thread need very little resources.

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 test program will produce the optimal number of threads per block for that specific GPU kernel.

a simple program similarities can be seen between CPU and GPU coding. The main difference between CPU and GPU coding is how computations are done. The CPU function on line 40 (defined on lines 5-8) is called once and it runs once. The GPU kernel on line 76 (defined on lines 10-18) is called once but it runs numBlocks$\times$numTreadsPerBlock times in parallel.

Each thread that executes the kernel is given a threadIdx and a blockDim with a blockIdx. Each of these integers are three dimensional, but only one dimension is used in this thesis. The integer threadIdx is the index of the thread inside the block. The integer blockDim is the number of threads per block launched. The integer blockIdx is the index of the block.

3.4 NVIDIA Telsa GPU architecture

A little bit of GPU architecture knowledge goes a long way. This thesis uses NVIDIA Tesla K40c and K20c GPUs, Figure 3.11 shows the form factor of the these GPUs. NVIDIA's line of Tesla GPUs are built specifically for general purpose computing rather than dedicated graphics processing. The red box in Figure 3.12 show the GPU chip and the yellow boxes show the SRAM that is *off* the GPU chip.

Figure 3.13 shows a block diagram of the Tesla architecture on the K20c and K40c GPUs. The global memory and L2 memory are located in the SRAM off chip. The GPU chip has thou-

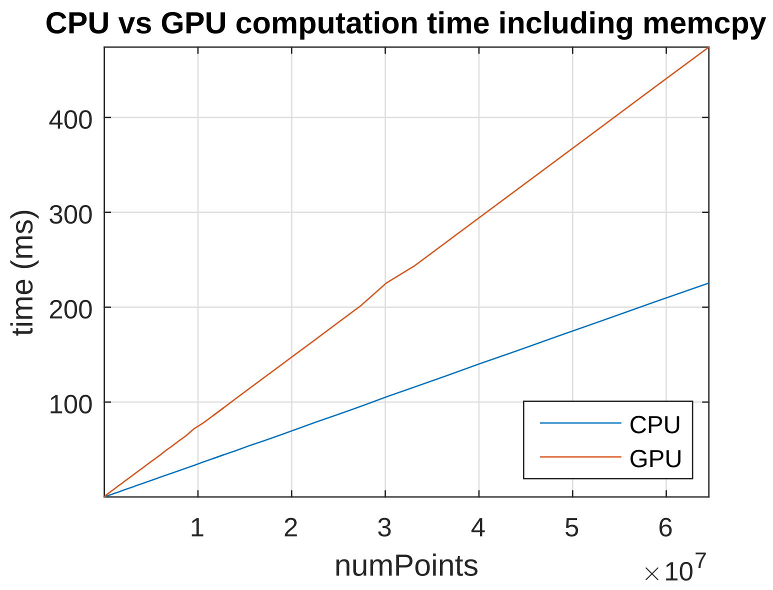


Figure 3.8: A comparison of CPU computation time verse GPU kernel computation time including memcpy from host to device and device to host.

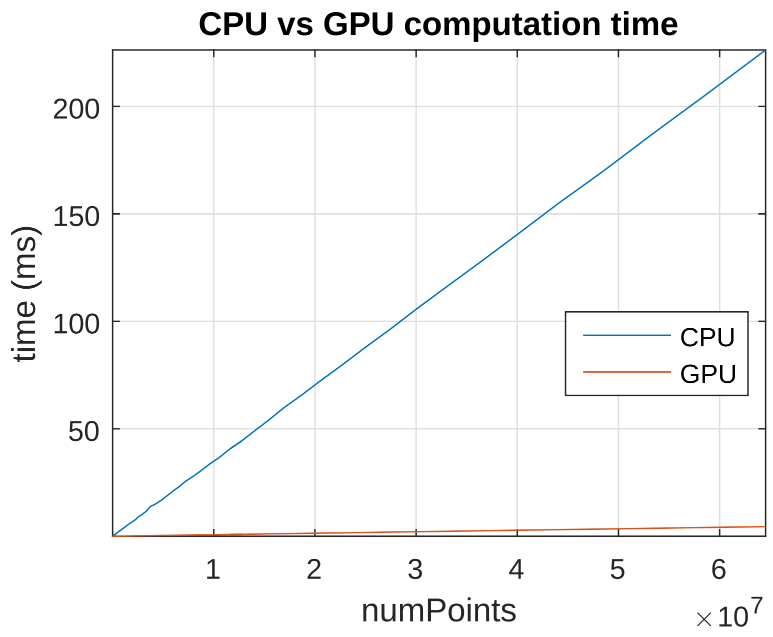


Figure 3.9: A comparison of CPU computation time verse GPU kernel computation.

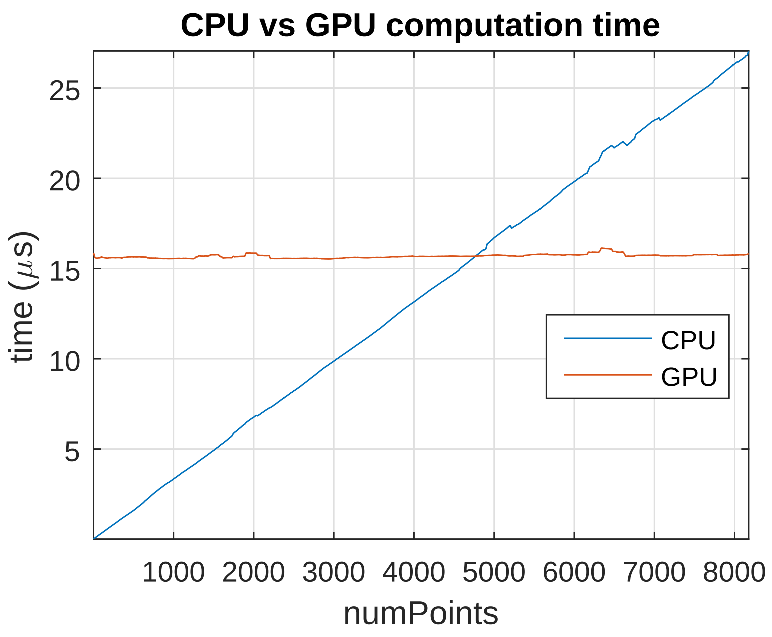


Figure 3.10: A comparison of CPU computation time verse GPU kernel computation. Note: time is in μs.

Figure 3.11: NVIDIA Tesla K40c and K20c.

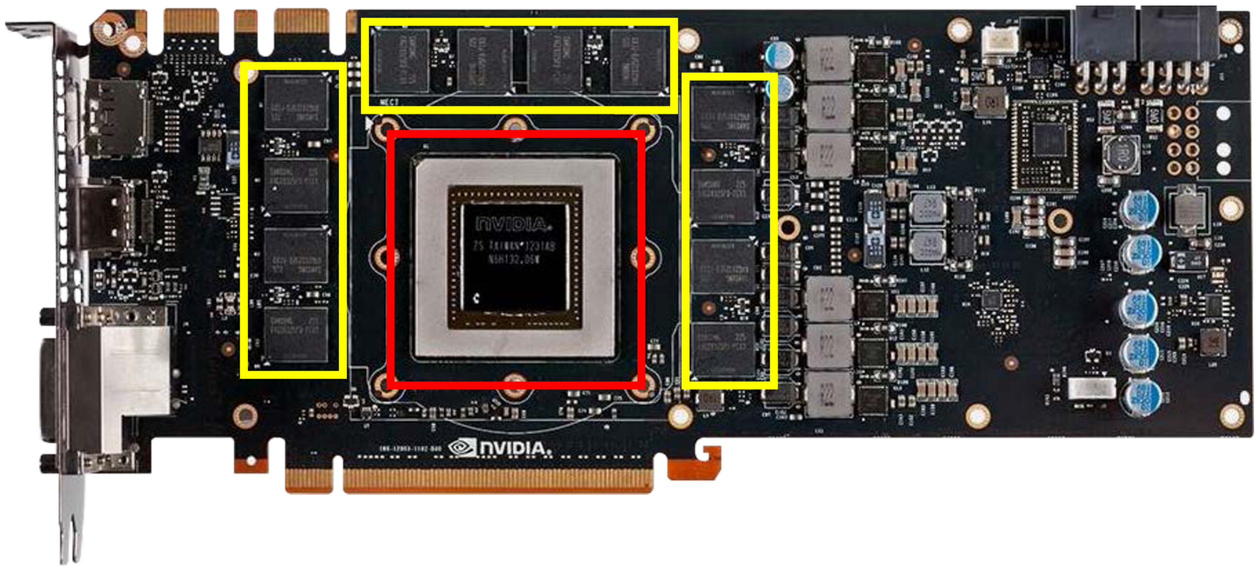


Figure 3.12: Example of an NVIDIA GPU card. The SRAM is shown to be boxed in yellow. The GPU chip is shown to be boxed in red.

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

Table 3.1: The computational resources available with three NVIDIA GPUs used in this thesis (1x Tesla K40c 2x Tesla K20c).

sands of relatively slow CUDA cores grouped together into streaming multiprocessors (SM). GPU threads are assigned to SMs where they have access to registers, L1 cache and shared memory. Table 3.1 summarizes the resources on the Tesla K40c and K20c.

The only memory that kernels have full control of is global memory and shared memory. Data from host memory is transferred over the PCI Express bus to global memory. Registers, L1 cache and L2 cache are handled and controlled by CUDA.

3.5 CUDA programming

A GPU cannot be programmed like a CPU because the architecture is totally different

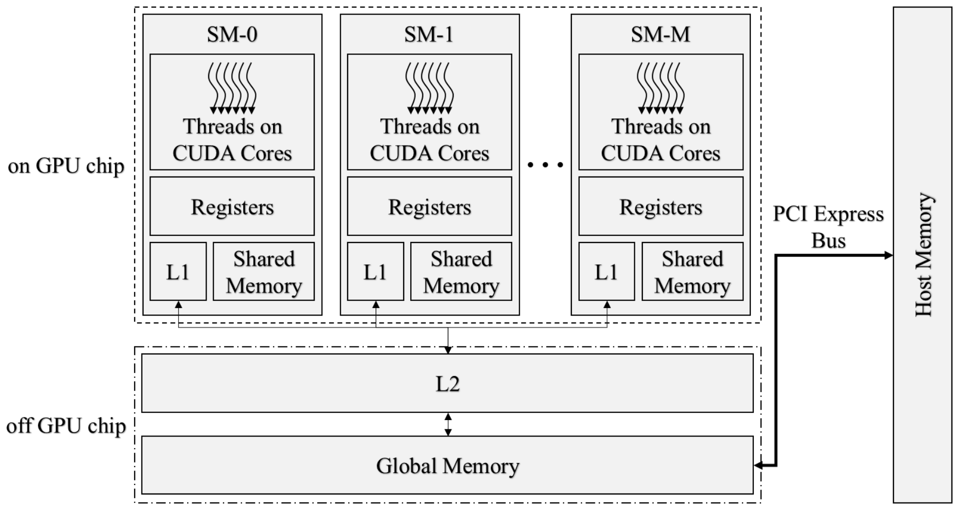


Figure 3.13: Block diagram of Tesla GPU microarchitecture. The dashed box indicates what blocks are located on the GPU chip. The dash dotted box indicates what blocks are located off the GPU chip.

When a kernel brings memory on chip from the off chip SRAM it takes 400-800 clock cycles. Accessing the SRAM memory is inevitable but kernels should be designed to access SRAM memory as little as possible.

CUDA

Kernels

Threads

Libraries

GPU memory

Bad things to do in CUDA

Good things to do in CUDA

Massaging algorithms to map to a GPU well

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$_{\text{hex}}$ and the ASM field

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

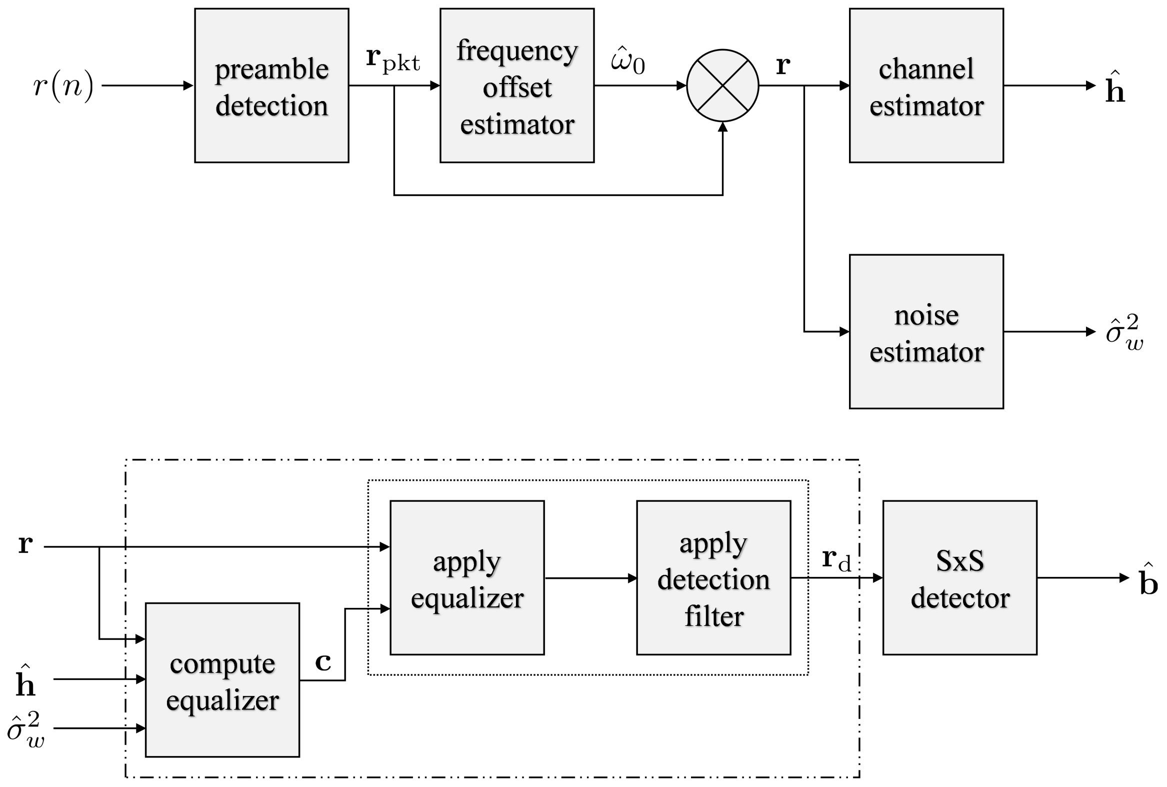


Figure 4.1: This a simple block diagram of what the GPU does.

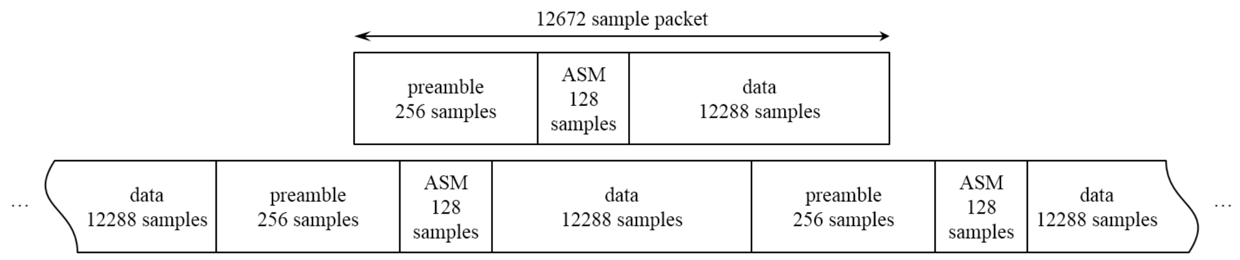


Figure 4.2: The iNET packet structure.

Each 16-bit sequence CD98$_{\text{hex}}$ 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 [4]. 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 $\mathbf{r}_{\text{pkt}}$.

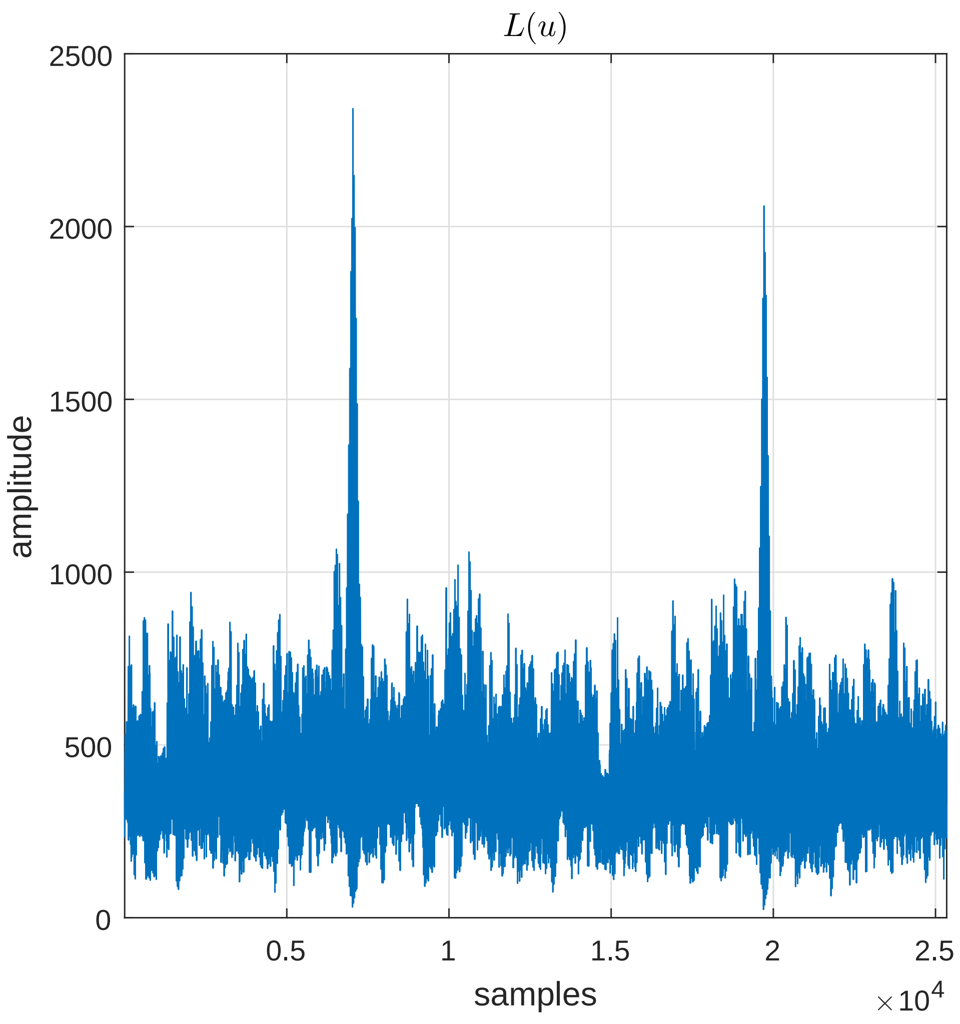


Figure 4.3: The output of the Preamble Detector $L(u)$.

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

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

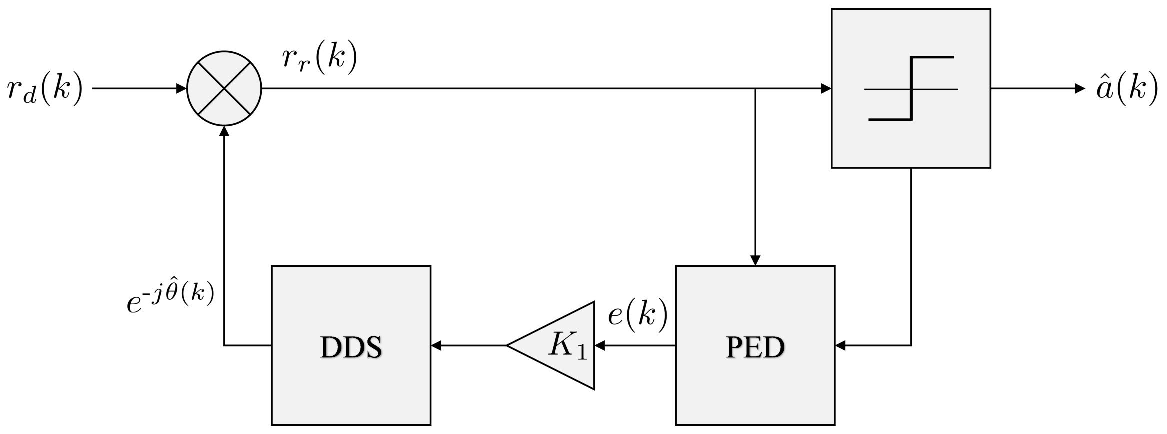
where ρ is the pre-computed constant

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

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

There are 3 different kinds of equalizers I run 1. the solving ones!!! They are equations like Ax=b where I have A and b but I need x 2. the initialized then iterative ones. CMA is initialized with MMSE then runs as many times a possible 3. the multiply ones! the FDEs are a simple multiply in the frequency domain

5.2 Zero-Forcing and MMSE Equalizers

The Zero-Forcing (ZF) and MMSE equalizers are treated together here because they have many common features. . .

The ZF equalizer is an FIR filter defined by the coefficients

$$c_{\text{ZF}}(-L_1) \quad \cdots \quad c_{\text{ZF}}(0) \quad \cdots \quad c_{\text{ZF}}(L_2). \tag{5.1}$$

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

$$\mathbf{H}\mathbf{c}_{\text{ZF}} = \mathbf{u}_{n_0} \tag{5.2}$$

where

$$\mathbf{c}_{\text{ZF}} = \begin{bmatrix} c_{\text{ZF}}(-L_1) \\ \vdots \\ c_{\text{ZF}}(0) \\ \vdots \\ c_{\text{ZF}}(L_2) \end{bmatrix}, \tag{5.3}$$

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} n_0 - 1 \text{ zeros} \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \end{matrix}, \tag{5.4}$$

and

$$\mathbf{H} = \begin{bmatrix} h(-N_1) & & & \\ h(-N_1+1) & h(-N_1) & & \\ \vdots & \vdots & \ddots & \\ h(N_2) & h(N_2-1) & & h(-N_1) \\ & h(N_2) & & h(-N_1+1) \\ & & & \vdots \\ & & & h(N_2) \end{bmatrix}. \tag{5.5}$$

Jeff explains how cuda solvers handle this equation.

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

$$\mathbf{c}_{\text{ZF}} = (\mathbf{H}^{\dagger}\mathbf{H})^{-1}\mathbf{H}^{\dagger}\mathbf{u}_{n_0} \tag{5.6}$$

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 $n0 = N_1 + L_1 + 1$

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \left.\vphantom{\begin{matrix}0\\ \vdots\\ 0\end{matrix}}\right\} n_0 - 1 \text{ zeros} \\ \vphantom{1} \\ \left.\vphantom{\begin{matrix}0\\ \vdots\\ 0\end{matrix}}\right\} N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \end{matrix} . \tag{5.7}$$

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

$$\mathbf{H} = \begin{bmatrix} h(-N_1) & & & \\ h(-N_1+1) & h(-N_1) & & \\ \vdots & \vdots & \ddots & \\ h(N_2) & h(N_2-1) & & h(-N_1) \\ & h(N_2) & & h(-N_1+1) \\ & & & \vdots \\ & & & h(N_2) \end{bmatrix} . \tag{5.8}$$

The computation of the coefficients in Equation (5.6) 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

$$\mathbf{R}_h = \mathbf{H}^\dagger\mathbf{H} = \begin{bmatrix} r_h(0) & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) \end{bmatrix} \tag{5.9}$$

where

$$r_h(k) = \sum_{n=-N_1}^{N_2} h(n)h^*(n-k). \tag{5.10}$$

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

$$\mathbf{h}_{n_0} = \mathbf{H}^\dagger \mathbf{u}_{n_0} = \begin{bmatrix} h(L_1) \\ \vdots \\ h(0) \\ \vdots \\ h(-L_2) \end{bmatrix}. \tag{5.11}$$

To simplify, Equations (5.9) and (5.11) are substituted into Equation (5.6) resulting in

$$\mathbf{c}_{\text{ZF}} = \mathbf{R}_h^{-1}\mathbf{h}_{n_0}. \tag{5.12}$$

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

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

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.

$$\mathbf{c}_{MMSE} = \left[\mathbf{G}\mathbf{G}^\dagger + \frac{\sigma_w^2}{\sigma_s^2}\mathbf{I}_{L_1+L_2+1}\right]\mathbf{g}^\dagger \tag{5.14}$$

where

$$\mathbf{G} = \begin{bmatrix} h(N_2) & \cdots & h(-N_1) & & & \\ & h(N_2) & \cdots & h(-N_1) & & \\ & & \ddots & & \ddots & \\ & & & h(N_2) & \cdots & h(-N_1) \end{bmatrix} \tag{5.15}$$

and

$$\mathbf{g} = \begin{bmatrix} h(L_1) \cdots h(-L_2) \end{bmatrix}. \tag{5.16}$$

The vector $\mathbf{g}^\dagger$ is also the same vector as $\mathbf{h}_{n_0}$ in Equation (5.7). The matrix multiplication $\mathbf{G}\mathbf{G}^\dagger$ is also the same autocorrelation matrix $\mathbf{R}_h$ as Equation (5.9). The fraction $\frac{1}{2\hat{\sigma}_w^2}$ 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 (5.14) results in

$$\left[\mathbf{R}_h + \frac{1}{2\hat{\sigma}_w^2}\mathbf{I}_{L_1+L_2+1}\right]\mathbf{c}_{\text{MMSE}} = \mathbf{h}_{n_0}. \tag{5.17}$$

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

$$\mathbf{R}_{hw} = \mathbf{R}_h + \frac{1}{2\hat{\sigma}_w^2}\mathbf{I}_{L_1+L_2+1} = \begin{bmatrix} r_h(0) + \frac{1}{2\hat{\sigma}_w^2} & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) + \frac{1}{2\hat{\sigma}_w^2} & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) + \frac{1}{2\hat{\sigma}_w^2} \end{bmatrix}. \tag{5.18}$$

The MMSE equalizer coefficients are solved for in a similar fashion to the Zero-Forcing equalizer coefficients in Equation (5.13).

$$\mathbf{R}_{hw}\mathbf{c}_{\text{MMSE}} = \mathbf{h}_{n_0}. \tag{5.19}$$

5.2.1 The Iterative Equalizer

The Constant Modulus Algorithm

CMA uses a steepest decent algorithm.

$$\mathbf{c}_{b+1} = \mathbf{c}_b - \mu\nabla\mathbf{J} \tag{5.20}$$

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

$$\nabla J = \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \Big[y(n)y^*(n) - R_2\Big]\, y(n)\mathbf{r}^*(n). \tag{5.21}$$

where

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

This means ∇J is of the form

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

To leverage the computational efficiency of the Fast Fourier Transform (FFT), Equation (5.21) is re-expressed as a convolution.

To begin messaging ∇J

$$z(n) = 2\Big[y(n)y^*(n) - R_2\Big]\, y(n) \tag{5.24}$$

is defined to make the expression of ∇J to be

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

then writing the summation out in vector form

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

The kth value of ∇J is

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

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

$$\rho(n) = r^*(n). \tag{5.28}$$

Now

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

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

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

$$= \sum_{m=0}^{L_{pkt}-1} z(m) r^*(m-n) \tag{5.30}$$

Comparing Equation (5.29) and (5.30) shows that

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

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

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

5.2.2 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

$$C_{\text{FDE1}}(\omega) = \frac{\hat{H}^*(\omega)}{|\hat{H}(\omega)|^2 + \frac{1}{\hat{\sigma}^2}} \tag{5.32}$$

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 (5.32) 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

$$C_{\text{FDE2}}(\omega) = \frac{\hat{H}^*(\omega)}{|\hat{H}(\omega)|^2 + \frac{\Psi(\omega)}{\hat{\sigma}^2}} \tag{5.33}$$

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 (5.33) is easily implemented in the GPU and is computationally efficient.

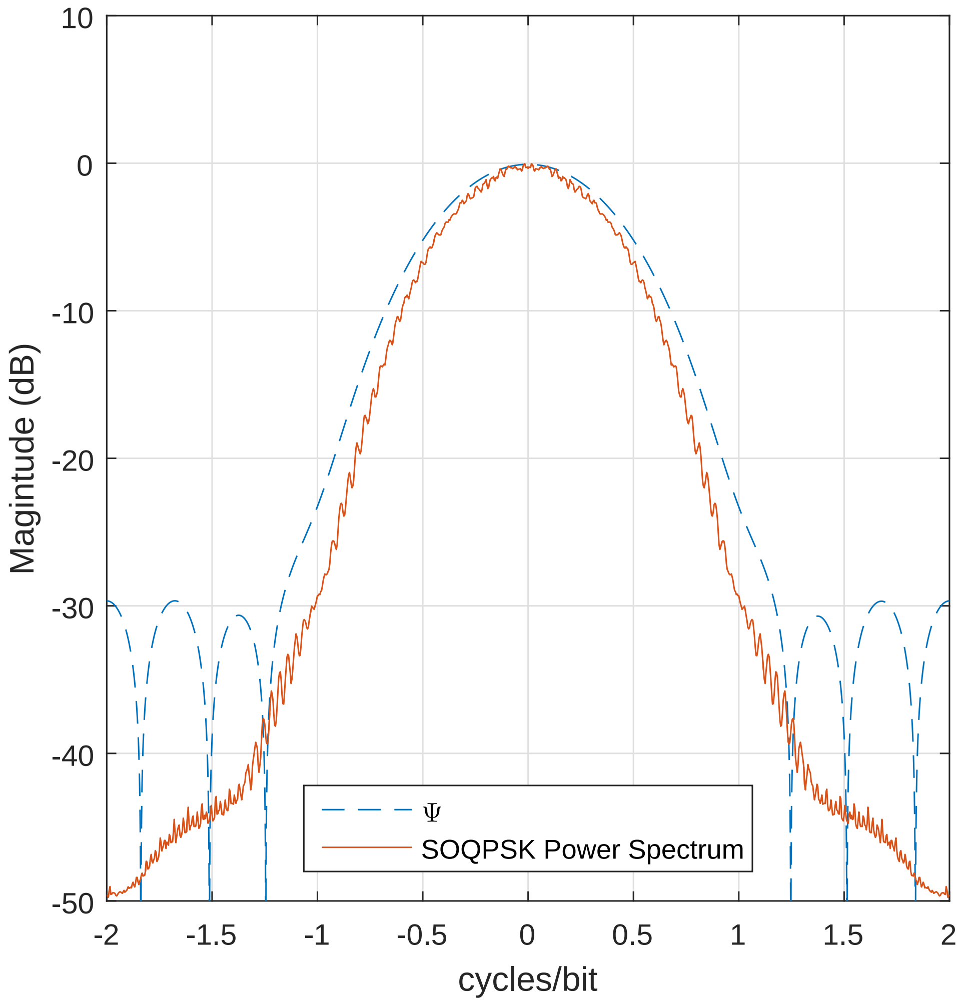


Figure 5.1: A block diagram illustrating organization of the algorithms in the GPU.

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 3, 7

[2] ——, “CUDA,” 2015. [Online]. Available: http://en.wikipedia.org/wiki/CUDA 4

[3] NVIDIA, “Cuda toolkit documentation,” 2017. [Online]. Available: http://docs.nvidia.com/cuda/ 7

[4] 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. 21