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

Signal Processing with GPUs

Frame Synchronization

To compute preamble assisted equalizers, estimators use the preamble to estimate various parameters. The iNET packet is comprised of three different sections: preamble, asynchronous marker (ASM) and the data. The packet and received sample structure is shown in Figure2.2.

The goal of the frame synchronization step is to synchronize the received samples into frames or packets by locating the preambles. To give an overview of the frame synchronization step, synchronization of the received samples is explained using a preamble detector and search algorithms to packetize the received samples. To find the preambles in the batch, a preamble detector is used to compute the sample correlation function between the received samples and the preamble. Search algorithms then search the correlation function for sample indices that correlate strongest to the preamble. These indices are the starting indices of each packet in the received samples. Finally, using the starting indices of each packet, received samples are structured and synchronized into frames or packets.

The first step in the frame synchronizer is to compute the sample correlation between the received samples and the preamble. A lower complexity preamble detector is shown in equation

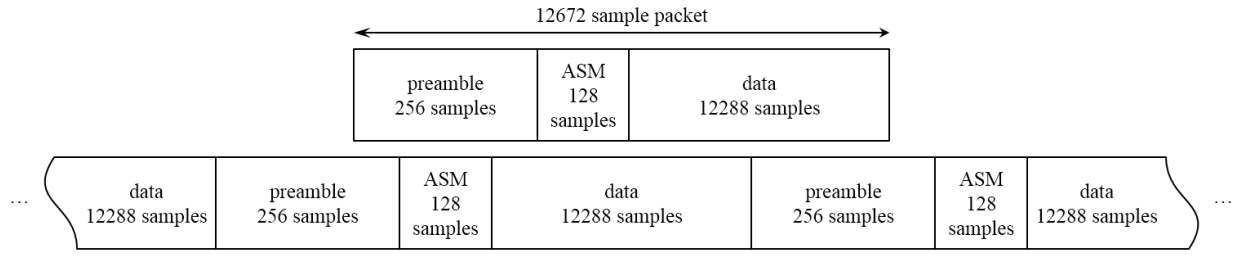

Figure 1.1: The iNET packet structure.

(**??**)-(**??**) in section **??** and repeated here for convenience.

$$L(n) = \sum_{m=0}^{7} \left[I^2(n,m) + Q^2(n,m) \right] \tag{1.1}$$

where

$$\begin{aligned} I(n,m) \approx & \sum_{\ell \in \mathcal{L}_1} r_R(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_2} r_R(\ell+32m+n) + \sum_{\ell \in \mathcal{L}_3} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_4} r_I(\ell+32m+n) \\ & + 0.7071 \left[\sum_{\ell \in \mathcal{L}_5} r_R(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_6} r_R(\ell+32m+n) \right. \\ & \left. + \sum_{\ell \in \mathcal{L}_7} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_8} r_I(\ell+32m+n) \right], \end{aligned} \tag{1.2}$$

$$\begin{aligned} Q(n,m) \approx & \sum_{\ell \in \mathcal{L}_1} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_2} r_I(\ell+32m+n) \\ & - \sum_{\ell \in \mathcal{L}_3} r_R(\ell+32m+n) + \sum_{\ell \in \mathcal{L}_4} r_R(\ell+32m+n) \\ & + 0.7071 \left[\sum_{\ell \in \mathcal{L}_5} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_6} r_I(\ell+32m+n) \right. \\ & \left. - \sum_{\ell \in \mathcal{L}_7} r_R(\ell+32m+n) + \sum_{\ell \in \mathcal{L}_8} r_R(\ell+32m+n) \right] \end{aligned} \tag{1.3}$$

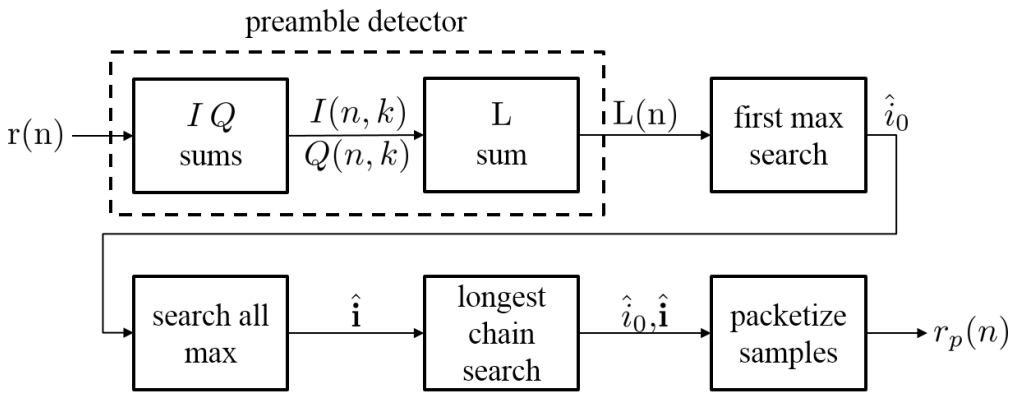

Figure 1.2: The block diagram for the frame synchronization implementation.

with

$$\begin{aligned}
\mathcal{L}_1 &= \{0, 8, 16, 24\} \\
\mathcal{L}_2 &= \{4, 20\} \\
\mathcal{L}_3 &= \{2, 10, 14, 22\} \\
\mathcal{L}_4 &= \{6, 18, 26, 30\} \\
\mathcal{L}_5 &= \{1, 7, 9, 15, 17, 23, 25, 31\} \\
\mathcal{L}_6 &= \{3, 5, 11, 12, 13, 19, 21, 27, 28, 29\} \\
\mathcal{L}_7 &= \{1, 3, 9, 11, 12, 13, 15, 21, 23\} \\
\mathcal{L}_8 &= \{5, 7, 17, 19, 25, 27, 28, 29, 31\}.
\end{aligned} \tag{1.4}$$

The preamble detector is implemented in the GPU in two kernels as shown by the dotted box in Figure 2.3. The first kernel computes the inner summations and the second computes the outer summation.

The inner summation, as defined in equations (2.2)-(2.4), computes an I and Q pair. A single I and Q pair is computed by summing 32 scaled real or imaginary parts of received samples. For each received sample an I and Q pair is computed.

The outer summation, as defined in equation (2.1), computes the reduced complexity maximum likelihood correlation L. One sample of L is computed by summing 8 squared then summed I and Q pairs. For each received sample L is computed.

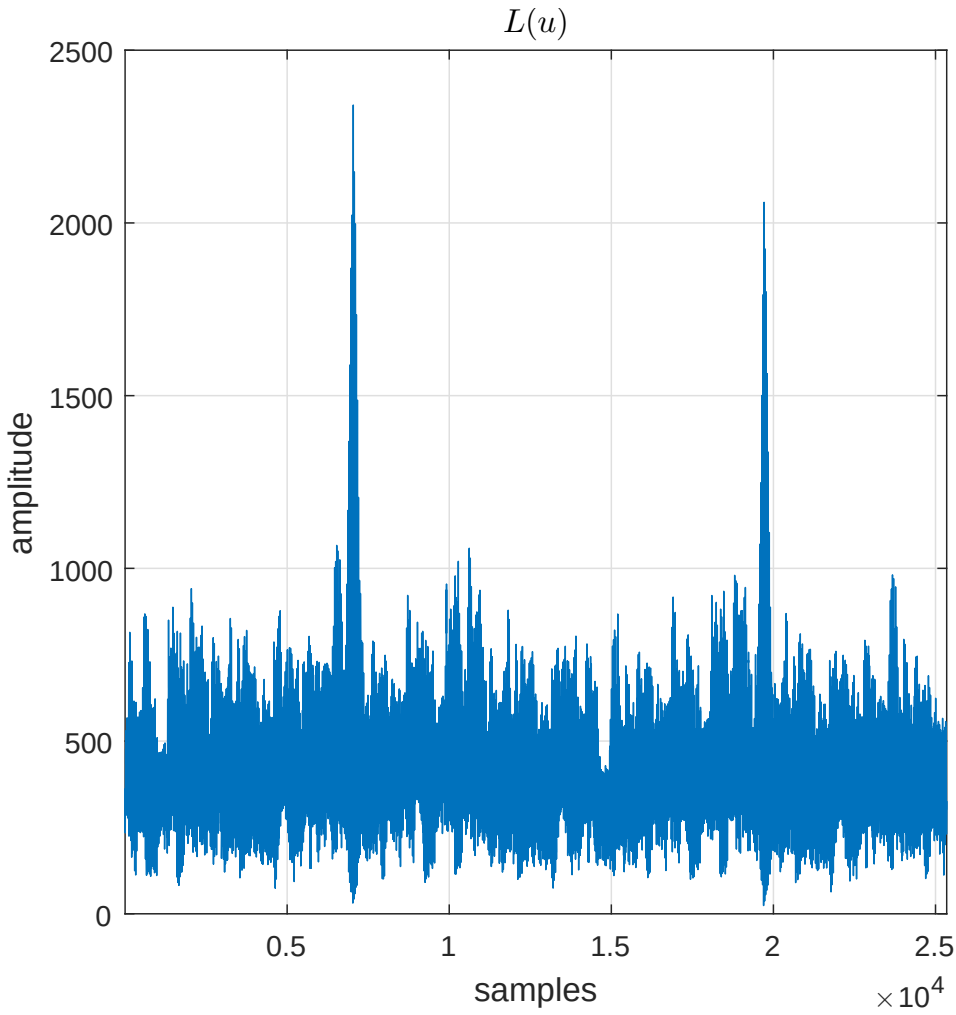

Figure 1.3: The output of the Preamble Detector $L(u)$.

Figure 2.4 shows an example of the first $2 \times L_{pkt}$ samples of L. The local maximums in L in the figure indicate a preamble starts at the maximum's sample index. The first local maximum is at sample index 7040, indicating the first packet starts at that index. The figure also shows the second packet starts at sample index 19712. The difference between these sample indices is L_{pkt}, this difference agrees with the packet length shown in Figure 2.2.

Because of the structure of the preamble, the preamble detector output L has some unique properties. The Figures in 2.5 show the correlation function around an expected preamble location. The correlation functions have peaks every 32 samples because the preamble bit sequence comprises eight repetitions of the 16-bit pattern CD98_{hex}. The repetitive structure causes one main correlation peak with seven side peaks.

When ideal samples are received the preamble detector output looks like Figure 2.5(a). The structure of the correlation peaks still occur when the signal to noise ratio is low, as shown in Figure 2.5(b). But when the signal to noise ratio is low and major multipath distortion happen, the correlation peaks look like Figures 2.5(c) and (d). The structure of the correlation peaks can cause a simple algorithm to find an incorrect preamble starting location.

In the worst case scenario, a simple algorithm might find an incorrect preamble index by searching a poorly placed search window. A poorly placed window might search the large side correlation peaks from Figure 2.5(a) and the small main correlation peaks from Figures 2.5(c) or (d). Because the first three side peaks in Figure 2.5(a) are much taller than the main peak in Figures 2.5(c) and (d), an incorrect preamble starting indices will result if search windows are not defined safely.

To prevent searching multiple preamble correlation peaks, search windows should only search the correlation peak of one preamble. To ensure the correlation peaks from only one preamble is searched, windows of length L_{pkt} are centered on expected preamble starting locations. Figure 2.6 shows an example of safe search windows centered on expected preamble starting locations.

To define safe search windows, a rough estimate of the preamble starting locations is made. The GPU launches a kernel with one thread to search for the maximum in the first L_{pkt} samples of the received samples. The maximum index is saved as $\hat{i}_0$. $\hat{i}_0$ is used to make a rough estimate for all the preamble staring locations by building a vector defined as

$$\hat{\mathbf{i}} = \begin{bmatrix} \hat{i}_0 + 0 \times L_{pkt} \\ \hat{i}_0 + 1 \times L_{pkt} \\ \vdots \\ \hat{i}_0 + 3102 \times L_{pkt} \\ \hat{i}_0 + 3103 \times L_{pkt} \end{bmatrix} \tag{1.5}$$

With rough estimates of where the preambles should be located, the GPU searches safe windows of L for local maximums. On local maximum is found in each search window by centering each window on elements from $\hat{\mathbf{i}}$. The GPU launches one thread per search window to find the

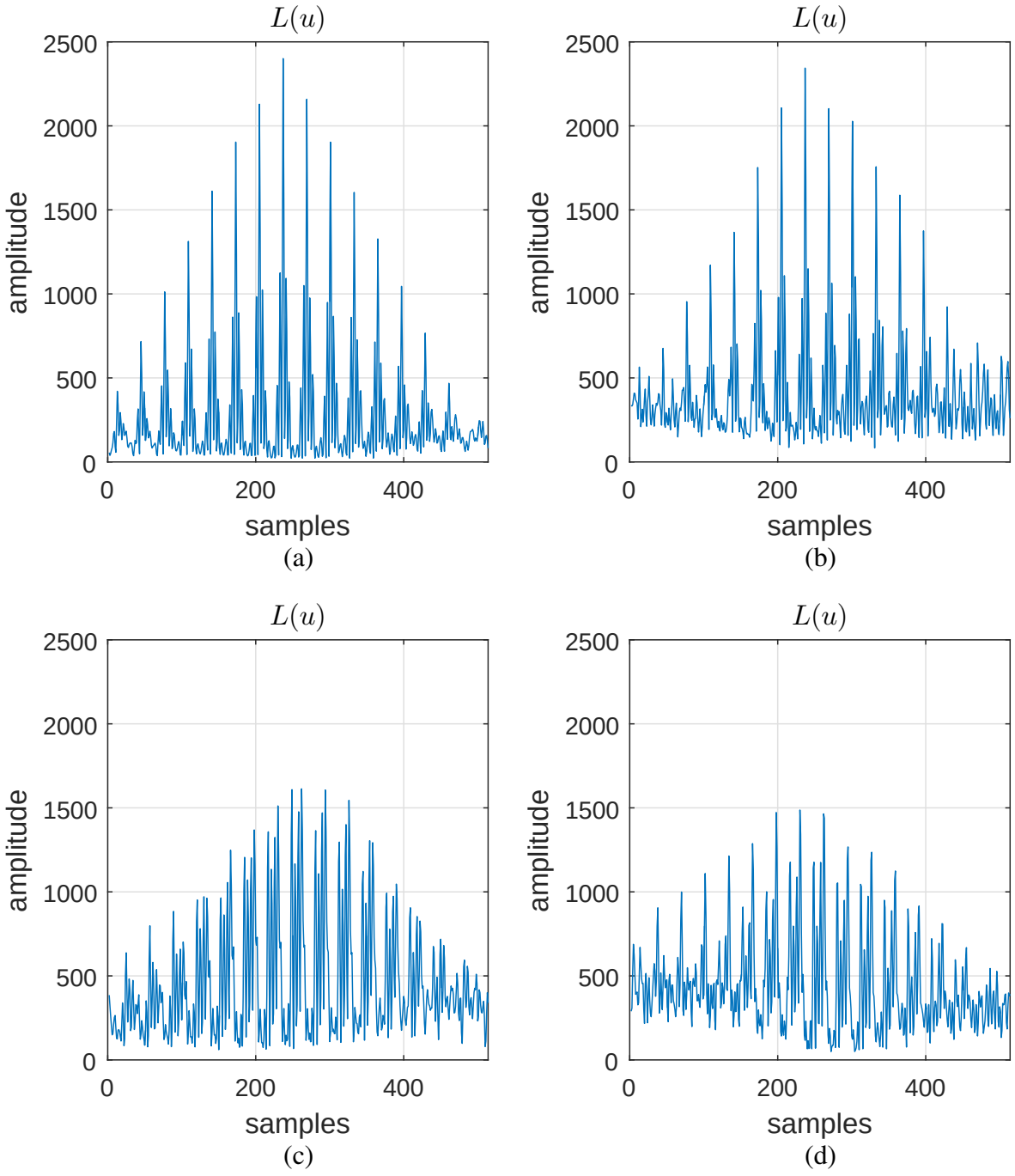

Figure 1.4: Detailed view of $L(u)$. (a): correlation peaks of a distortion free and noiseless signal; (b): correlation peaks of a distortion free but noisy signal with $E_b/N_0 = 0$dB; (c): correlation peaks of a distorted and noisy signal with $E_b/N_0 = 0$dB; (d): correlation peaks of a distorted and noisy signal with $E_b/N_0 = 0$dB

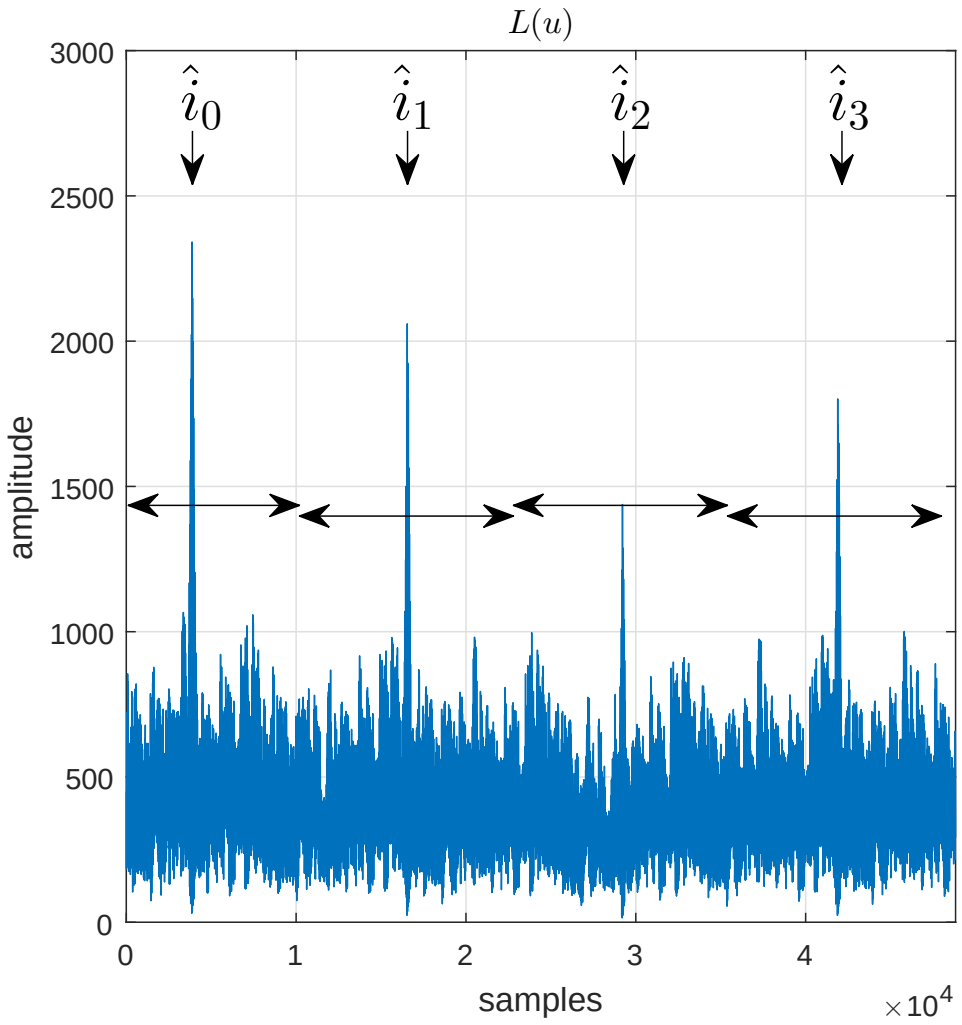

Figure 1.5: Safe search windows defined to search only one preamble correlation peak.

local maximum and save its sample index in $\hat{\mathbf{i}}$. The vector $\hat{\mathbf{i}}$ has the sample indices for 3103 local maximums that should be L_{pkt} samples apart.

Because of noise and multipath, the sample indices in $\hat{\mathbf{i}}$ are not the ideal L_{pkt} samples. The GPU launches one thread to search $\hat{\mathbf{i}}$ for the longest chain of perfectly spaced indices. The modulo of the last index in the longest chain of indices spaced L_{pkt} samples apart is the best estimate for $\hat{i}_0$. Once again, $\hat{i}_0$ is updated with the best estimated first preamble starting location. The vector $\hat{\mathbf{i}}$ is also updated again as in equation (2.5).

Now with the best estimate of the preamble starting locations, the received samples can be packetized and synchronized. Figure 2.7 shows the relationship of $\hat{\mathbf{i}}$ and $\boldsymbol{r}$. Using $\hat{\mathbf{i}}$, the GPU launches one thread per received sample to packetize $\boldsymbol{r}$ into $\boldsymbol{r}_p$ as shown in Figure 2.8. The

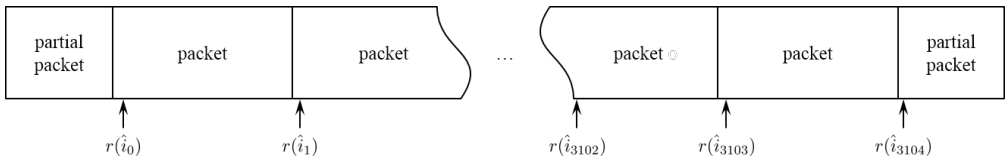

Figure 1.6: The starting sample index for each packet in the batch.

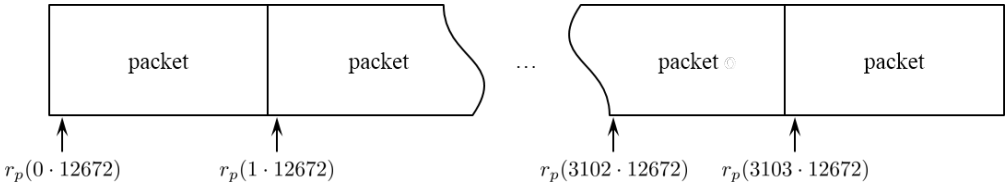

Figure 1.7: The packetized structure of the received signals after the frame synchronization step.

structure of r_p majorly simplifies indexing and removes the need for $\hat{\mathbf{i}}$.

$$\boldsymbol{r}_p = \begin{bmatrix} r(\hat{i}_0) \\ r(\hat{i}_0 + 1) \\ \vdots \\ r(\hat{i}_0 + 12670) \\ r(\hat{i}_0 + 12671) \\ r(\hat{i}_1) \\ r(\hat{i}_1 + 1) \\ \vdots \\ r(\hat{i}_{3103} + 12670) \\ r(\hat{i}_{3103} + 12671) \end{bmatrix} \tag{1.6}$$

The steps in Figure 2.3 synchronize the received samples r into r_p by using a preamble detector. The output of the preamble detector, L, is searched for local maximums. Imperfect spacing of local maximums is fixed by finding the longest chain of perfectly spaced indices in $\hat{\mathbf{i}}$. The received samples are then packetized based on the corrected preamble spacing.

Frequency Offset Compensation

As discussed in section **??**, all the estimators depend on atleast one of the other parameters. Estimating the frequency offset using the periodic properties of the preamble gives an accurate estimate, even in multipath. The frequency offset is estimated using the data-aided frequency offset estimator in equation (**??**). To compensate for the frequency offset, the received samples are rotated by the negative of estimated frequency offset. The data-aided frequency offset estimator equation is repeated here for convenience.

$$\hat{\omega}_0 = \frac{1}{L_q} \arg \left\{ \sum_{n=i+2L_q}^{i+7L_q-1} r(n) r^*(n - L_q) \right\} \tag{1.7}$$

where $\hat{\omega}_0$ is the frequency offset estimate for a single packet. The summation in equation (1.7) can be reformulated in terms of an inner product

$$\hat{\omega}_0 = \frac{1}{L_q} \arg \left\{ \mathbf{r}_{p1}^T \mathbf{r}_{p2}^* \right\}. \tag{1.8}$$

where

$$\mathbf{r}_{p1} = \begin{bmatrix} r_p(2L_q) \\ r_p(2L_q + 1) \\ \vdots \\ r_p(7L_q - 2) \\ r_p(7L_q - 1) \end{bmatrix} \tag{1.9}$$

$$\mathbf{r}_{p2}^* = \begin{bmatrix} r_p^*(L_q) \\ r_p^*(L_q + 1) \\ \vdots \\ r_p^*(6L_q - 2) \\ r_p^*(6L_q - 1) \end{bmatrix} \tag{1.10}$$

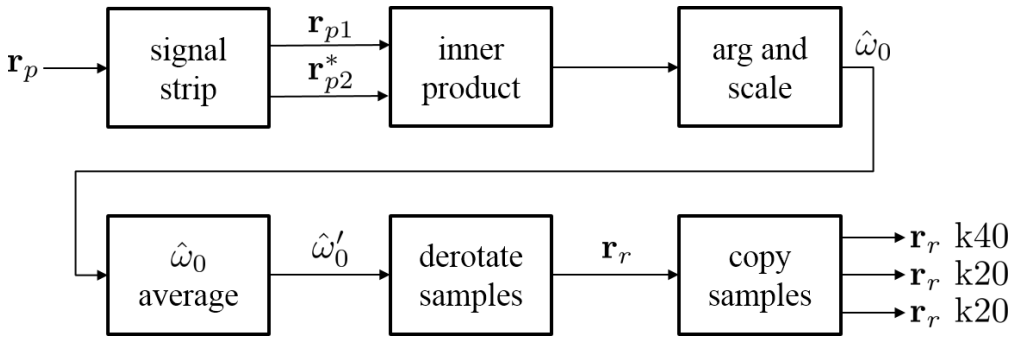

Figure 1.8: The block diagram for the Frequency Offset estimator and sample rotation implementation.

Reformulating the estimator into an inner product isn't only simpler, it is also much faster in the GPU. The GPU uses an extremely fast and efficient CUDA Basic Linear Algebra Subprogram (CUBLAS) library to compute the inner product [**?**]. Figure 1.8 shows how equation (1.8) is implemented in GPU kernels. The GPU strips $\mathbf{r}_{p1}$ and $\mathbf{r}^*_{p2}$ from the received packetized samples by launching one thread per packetized sample.

The GPU computes the inner product of $\mathbf{r}_{p1}$ and $\mathbf{r}^*_{p2}$ using a batched CUBLAS operation. The arg and scale GPU kernel launches one thread per packet to compute the scaled angle of the inner product for each packet. The scaled angle of each packet is saved in the vector $\hat{\boldsymbol{\omega}}_0$.

In aeronautical telemetry, the frequency offset can be assumed to be constant over two seconds. Because of this assumption, one frequency offset estimate is obtained by averaging all the offsets in a batch. The GPU launches one thread to compute the average of the vector $\hat{\boldsymbol{\omega}}_0$ and save the average in the scalar value $\hat{\omega}'_0$.

The averaged frequency offset $\hat{\omega}'_0$ is used to compensate for the frequency offset. The frequency offset compensation can be done with the matrix $\boldsymbol{\Omega}_r$ the rotations due to the frequency offset are in matrix notation using

$$\boldsymbol{r}_r = \boldsymbol{\Omega}_r \boldsymbol{r}_p = \begin{bmatrix} e^{-j\hat{\omega}'_0(0)} & & \\ & \ddots & \\ & & e^{-j\hat{\omega}'_0(3103\times L_{pkt}-1)} \end{bmatrix} \begin{bmatrix} r_p(0) \\ \vdots \\ r_p(3103 \times L_{pkt} - 1) \end{bmatrix}. \tag{1.11}$$

Implementing equation (1.11) in GPUs would use an extreme amount of memory and computation because of the size and dimension of $\boldsymbol{\Omega}_r$. Instead, the GPU launches one thread per

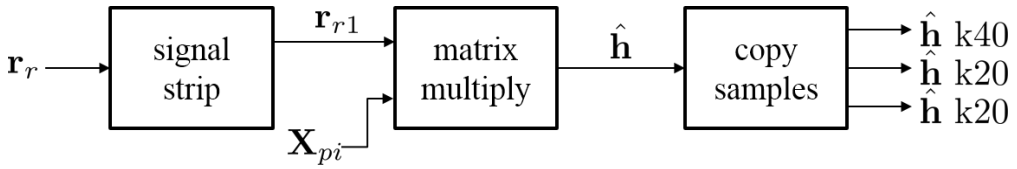

Figure 1.9: The block diagram for the Channel estimator.

packetized sample to rotate the packetized sample by $-\hat{\omega}_0'$.

$$r_r(n) = r_p(n) \times e^{-j\hat{\omega}_0' n} \tag{1.12}$$

With the frequency offset compensation done, the samples are transferred over the PCIe bus from the Tesla k40 GPU to the two Tesla k20 GPUs, shown as the last block in Figure 1.8.

Channel Estimation

As mentioned in **??**, the channel cannot be estimated with a frequency offset in the received samples. With the offset removed, the channel can accurately be estimated. Equation (**??**) is used to compute the channel estimate. The equation is repeated below for convenience.

$$\hat{\mathbf{h}} = \left(\mathbf{X}^\dagger \mathbf{X}\right)^{-1} \mathbf{X}^\dagger \hat{\mathbf{\Omega}}_0^\dagger \mathbf{r} \tag{1.13}$$

The channel estimator was originally formulated assuming the received samples still had a frequency offset. The samples in the GPU have already compensated for the offset and (1.13) is reformed into

$$\hat{\mathbf{h}} = \left(\mathbf{X}^\dagger \mathbf{X}\right)^{-1} \mathbf{X}^\dagger \mathbf{r}_{r1}. \tag{1.14}$$

where

$$\mathbf{r}_{r1} = \begin{bmatrix} r_r(N_2) \\ r_r(N_2+1) \\ \vdots \\ r_r(L_p - N_1 - 2) \\ r_r(L_p - N_1 - 1) \end{bmatrix} \tag{1.15}$$

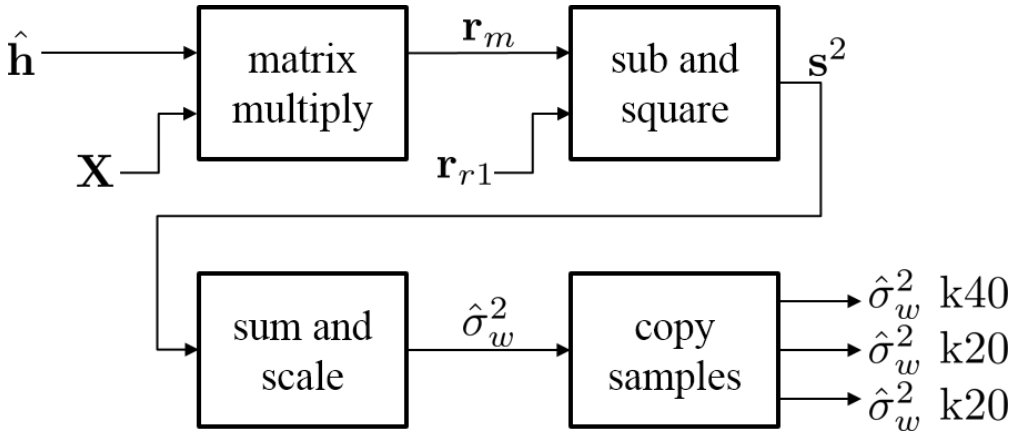

Figure 1.10: The block diagram for the Noise Variance estimator.

The matrix $\mathbf{X}$ is the $(L_p + L_a - N_1 - N_2) \times (N_1 + N_2 + 1)$ convolution matrix formed from the iNet preamble and ASM samples. The psnudo inverse of $\mathbf{X}$, $\left(\mathbf{X}^\dagger\mathbf{X}\right)^{-1}\mathbf{X}^\dagger$ is computed once and saved as $\mathbf{X}_{pi}$. Equipped with $\mathbf{X}_{pi}$ the channel estimator can be reformed again into

$$\hat{\mathbf{h}} = \mathbf{X}_{pi}\mathbf{r}_{r1}. \tag{1.16}$$

The GPU only needs compute a matrix operation after stripping a portion of $\mathbf{r}_r$. The GPU launches $L_p - N_1 - N_2$ threads per packet strip $\mathbf{r}_{r1}$ from $\mathbf{r}_r$. With $\mathbf{r}_{r1}$ built, the CUBLAS library is used to compute the matrix multiply to estimate the channel $\hat{\mathbf{h}}$. The channel estimates are then transferred from the k40 GPU to the k20 GPUs. Figure 1.9 shows the block diagram of how equation (1.16) is implemented in the GPU.

Noise Variance Estimation

With the frequency offset removed and the channel estimated, the noise variance can be estimated. The noise variance was studied in section **??** and **??**. Equation (**??**) is the final form of the noise variance estimator and repeated here for convenience.

$$\hat{\sigma}_w^2 = \frac{1}{2\rho}\left|\mathbf{r} - \hat{\mathbf{\Omega}}_0\mathbf{X}\hat{\mathbf{h}}\right|^2 \tag{1.17}$$

where

$$\rho = \text{Trace}\left\{\mathbf{I} - \mathbf{X}\left(\mathbf{X}^\dagger\mathbf{X}\right)^{-1}\mathbf{X}^\dagger\right\}. \tag{1.18}$$

Once again, $\mathbf{\Omega}_0$ is included in (1.17) assuming the vector $\mathbf{r}$ has a frequency offset. The vector $\mathbf{r}_{r1}$ are received samples with the frequency offset compensated for, $\mathbf{r}_{r1}$ is the same vector from the channel estimator. The noise variance estimator is reformulated replacing $\mathbf{r}$ and removing $\mathbf{\Omega}_0$.

$$\hat{\sigma}_w^2 = \frac{1}{2\rho} \left| \mathbf{r}_{r1} - \mathbf{X}\hat{\mathbf{h}} \right|^2 . \tag{1.19}$$

To begin estimating the noise variance, the GPU first computes the matrix multiplication $\mathbf{X}\hat{\mathbf{h}}$. The stored matrix $\mathbf{X}$ is a $(L_p + L_a - N_1 - N_2) \times (N_1 + N_2 + 1)$ convolution matrix and the channel estimate $\hat{\mathbf{h}}$ is a $(N_1 + N_2 + 1) \times 1$ vector. The result of the matrix product $\mathbf{r}_m$ is simply the ideal preamble distorted by the estimated multipath channel. The GPU calls a CUBLAS function to compute $\mathbf{r}_m$ quickly and efficiently.

With $\mathbf{r}_m$ computed, $\frac{1}{2\rho} |\mathbf{r}_{r1} - \mathbf{r}_m|^2$ is split into two GPU kernels. The first GPU kernel computes $\mathbf{s}^2$, an element by element magnitude squared difference vector of $\mathbf{r}_m$ and $\mathbf{r}_{r1}$ with $L_p + L_a - N_1 - N_2$ threads for each packet.

$$s^2(n) = |r_{r1}(n) - r_m(n)|^2 \tag{1.20}$$

For each packet, $\hat{\sigma}_w^2$ is computed by scaling the summation of by $\frac{1}{2\rho}$ in one GPU kernel. The GPU kernel is launched with one thread per packet to scale a summation of $\mathbf{s}^2$ by $1/2\rho$. ρ is a precomputed scalar based on the convolution matrix $\mathbf{X}$. With the noise variance estimated for each packet, the estimates are transferred from the k40 GPU to the k20 GPUs. These kernels are shown in Figure 1.10.

1.0.1 Equalizer GPU Implementation

Zero-Forcing Equalizer

The Zero-Forcing (ZF) equalizer was explored in section **??**. Equation (**??**) computes the ZF filter coefficients. The equation is repeated here for convenience with an added subscript.

$$\mathbf{c}_{ZF} = \left(\mathbf{H}^\dagger \mathbf{H}\right)^{-1} \mathbf{H}^\dagger \mathbf{u}_{n_0} \tag{1.21}$$

where $\mathbf{H}$ is the $(N_1+N_2+L_1+L_2+1)\times(L_1+L_2+1)$ convolution matrix formed by the channel impulse response $\hat{\mathbf{h}}$:

$$\mathbf{H} = \begin{bmatrix} \hat{h}(-N_1) & & & \\ \hat{h}(-N_1+1) & \hat{h}(-N_1) & & \\ \vdots & \vdots & \ddots & \\ \hat{h}(N_2) & \hat{h}(N_2-1) & & \hat{h}(-N_1) \\ & \hat{h}(N_2) & & \hat{h}(-N_1+1) \\ & & & \vdots \\ & & & \hat{h}(N_2) \end{bmatrix}. \tag{1.22}$$

The length L_1+L_2+1 is the length of the equalizer L_{eq}. The length N_1+B_2 is the length of the channel $L_c h$. and we assume $c(n)$ has support on $-L_1 \le n \le L_2$ and that $\hat{h}(n)$ has support on $-N_1 \le n \le N_2$. $\mathbf{u}_{n_0}$ is the $(N_1+N_2+L_1+L_2+1)\times 1$ vector representing the desired composite impulse response: that is, a vector comprising all-zeros with a one in position n_0. These vectors are

$$\mathbf{c}_{\text{ZF}} = \begin{bmatrix} c(-L_1) \\ \vdots \\ c(0) \\ \vdots \\ c(L_2) \end{bmatrix} \qquad \mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} n_0 - 1 \text{ zeros} \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} N_1+N_2+L_1+L_2-n_0+1 \text{ zeros} \end{matrix} \tag{1.23}$$

In equation (1.21), $\mathbf{H}^\dagger\mathbf{H}$ is of the form

$$\mathbf{R}_{h_{\text{ZF}}} = \mathbf{H}^\dagger\mathbf{H} = \begin{bmatrix} r_{h_{\text{ZF}}}(0) & r_{h_{\text{ZF}}}(-1) & \cdots & r_{h_{\text{ZF}}}(-L_1-L_2) \\ r_{h_{\text{ZF}}}(1) & r_{h_{\text{ZF}}}(0) & \cdots & r_{h_{\text{ZF}}}(-L_1-L_2+1) \\ \vdots & \vdots & \ddots & \vdots \\ \\ r_{h_{\text{ZF}}}(L_1+L_2) & r_{h_{\text{ZF}}}(L_1+L_2) & \cdots & r_{h_{\text{ZF}}}(0) \end{bmatrix} \tag{1.24}$$

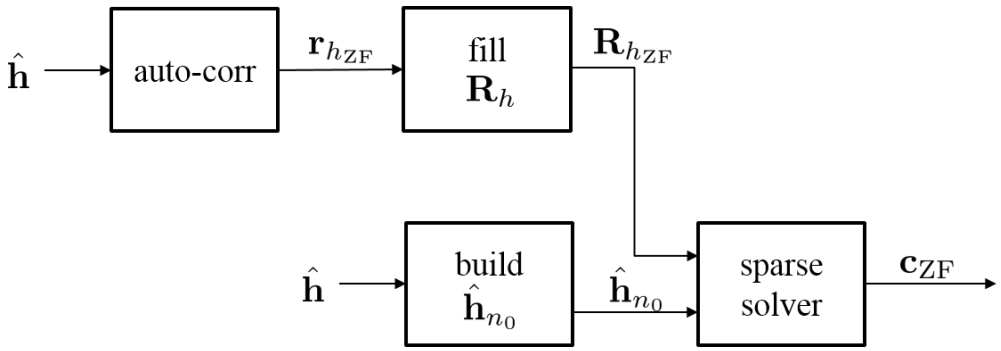

Figure 1.11: The block diagram for the Zero Forcing equalizer implementation.

where $r_{h_{\mathrm{ZF}}}(k)$ is the auto-correlation of the estimated channel

$$r_{h_{\mathrm{ZF}}}(k) = \sum_{n=-N_1}^{N_2} \hat{h}(n)\hat{h}^*(n-k). \tag{1.25}$$

The definition of $\mathbf{R}_{h_{\mathrm{ZF}}}$, requires $r_{h_{\mathrm{ZF}}}(k)$ for $-L_1 - L_2 \leq n \leq L_1 + L_2$. But $r_{h_{\mathrm{ZF}}}(k)$ only has support on $-N_1 - N_2 \leq n \leq N_1 + N_2$ resulting in $r_{h_{\mathrm{ZF}}}(k)$ being sparse which means $r_{h_{\mathrm{ZF}}}(k)$ is mostly zero. In (1.24), $r_{h_{\mathrm{ZF}}}(k)$ for $|k| > N_1 + N_2$ is zero resulting in $\mathbf{R}_{h_{\mathrm{ZF}}}$ also being sparse. In fact, $\mathbf{R}_{h_{\mathrm{ZF}}}$ is $\%64$ zeros.

In equation (1.21), $\mathbf{H}^\dagger \mathbf{u}_{n_0}$ can be simplified to the vector $\hat{\mathbf{h}}_{n_0}$

$$\mathbf{H}^\dagger \mathbf{u}_{n_0} = \hat{\mathbf{h}}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \mathbf{h} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} n_0 - N_1 - 1 \text{ zeros} \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} N_1 + L_1 + L_2 - n_0 + 1 \text{ zeros.} \end{matrix} \tag{1.26}$$

where $\hat{\mathbf{h}}_{n_0}$ is just a zero padded version of the channel estimate $\mathbf{h}$.

The computation of $\mathbf{c}_{\mathrm{ZF}}$ looks fairly straight forward but the inverse of $\mathbf{H}^\dagger\mathbf{H}$ presents a very challenging problem in GPU implementation. GPUs perform extremely well on parallel algorithms but an inverse is inherently serial. If the inverse was to implemented, the computation takes longer

than 1.9065 seconds. Equation (1.21) must be reformed to remove the inverse of $\mathbf{H}^{\dagger}\mathbf{H}$.

$$\mathbf{H}^{\dagger}\mathbf{H}\mathbf{c}_{\text{ZF}} = \mathbf{H}^{\dagger}\mathbf{u}_{n_0} \tag{1.27}$$

or

$$\mathbf{R}_{h_{\text{ZF}}}\mathbf{c}_{\text{ZF}} = \hat{\mathbf{h}}_{n_0} \tag{1.28}$$

To eliminate the need for an inverse, $\mathbf{H}^{\dagger}\mathbf{H}$ is moved to the left side of equation (1.21). Solving for $\mathbf{c}_{\text{ZF}}$ in (1.28) can be done many ways but the sparseness of $\mathbf{R}_{h_{\text{ZF}}}$ must be leveraged to ensure the coeffiencts can be calcualted with in the 1.9065 seconds. Nvidia has a batched sparse solver library that computes 3103 $\mathbf{c}_{\text{ZF}}$ equalizers in less than 410ms.

Figure 1.11 shows the ZF filter calculations only require the channel estimate $\hat{\mathbf{h}}$. The GPU launches a kernel with $2(N_1 + N_2) + 1$ threads per packet to calculate the estimated channel auto-correlation $\mathbf{R}_{h_{\text{ZF}}}$. The matrix $\mathbf{R}_{h_{\text{ZF}}}$ is built by the GPU with $\mathbf{R}_{h_{\text{ZF}}}$ by launching a kernel with a thread for every non-zero index in $\mathbf{R}_{h_{\text{ZF}}}$. The GPU also builds the vector $\hat{\mathbf{h}}_{n_0}$ by zero padding the channel estimate with $N_1 + N_2 + 1$ threads per packet. The Zero forcing equalizer coefficients are then calculated using the batched sparse solver from the CUDA libary.

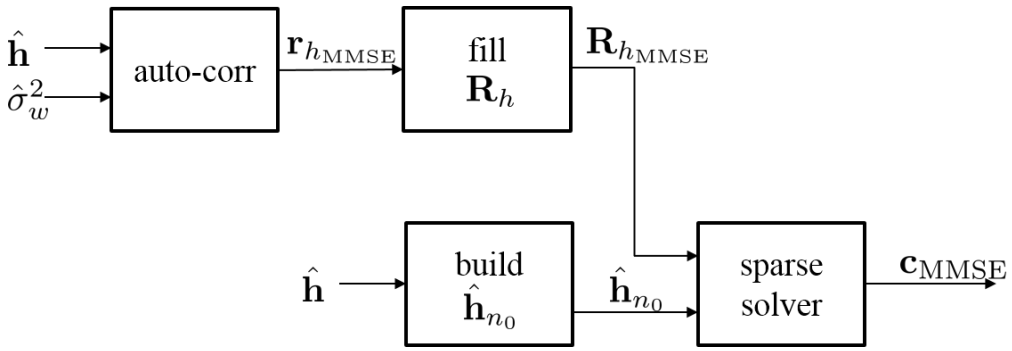

Figure 1.12: The block diagram for the MMSE equalizer implementation.

MMSE Equalizer

The general formulation for the MMSE equalizer is the same as ZF equalizer. The MMSE equalizer was explored in section **??**. Equation (**??**) computes the MMSE filter coefficients. The equation is repeated here for convenience with an added subscript. The MMSE optimum equalizer filter coefficients are given by

$$\mathbf{c}_{\text{MMSE}} = \left[\mathbf{H}^{\dagger}\mathbf{H} + \frac{2\sigma_w^2}{\sigma_s^2}\mathbf{I}_{L_1+L_2+1}\right]^{-1}\mathbf{g}^{\dagger} \tag{1.29}$$

where $\mathbf{H}$ is the same $\mathbf{H}$ from the ZF equalizer in equation (1.22) and $\mathbf{I}_{L_1+L_2+1}$ is the $(L_1+L_2+1)\times(L_1+L_2+1)$ identity matrix. The vector $\mathbf{g}^{\dagger}$ is also the same as the vector $\hat{\mathbf{h}}_{n_0}$ in the ZF equalizer in equation (1.26). The matrix operations in (1.29) can be expressed as

$$\left[\mathbf{H}^{\dagger}\mathbf{H} + \frac{2\sigma_w^2}{\sigma_s^2}\mathbf{I}_{L_1+L_2+1}\right] = \mathbf{R}_{h_{\text{MMSE}}} \tag{1.30}$$

where

$$\mathbf{R}_{h_{\text{MMSE}}} = \begin{bmatrix} r_{h_{\text{MMSE}}}(0) & r_{h_{\text{MMSE}}}(-1) & \cdots & r_{h_{\text{MMSE}}}(-L_1-L_2) \\ r_{h_{\text{MMSE}}}(1) & r_{h_{\text{MMSE}}}(0) & \cdots & r_{h_{\text{MMSE}}}(-L_1-L_2+1) \\ \vdots & \vdots & \ddots & \vdots \\ r_{h_{\text{MMSE}}}(L_1+L_2) & r_{h_{\text{MMSE}}}(L_1+L_2) & \cdots & r_{h_{\text{MMSE}}}(0) \end{bmatrix} \tag{1.31}$$

and

$$r_{h_{\text{MMSE}}}(k) = \sum_{n=-N_1}^{N_2} \hat{h}(n)\hat{h}^*(n-k) + \hat{\sigma}_w^2\delta(k). \tag{1.32}$$

The MMSE equalizer coeffiencts are calculated almost the same way as the ZF equalizer. The only little difference is calculating $r_{h_{\text{MMSE}}}(k)$, the noise variance $\hat{\sigma}_w^2$ is added to $r_{h_{\text{MMSE}}}(0)$. Aside from adding $\hat{\sigma}_w^2$, the GPU implements

$$\mathbf{R}_{h_{\text{MMSE}}}\mathbf{c}_{\text{MMSE}} = \hat{\mathbf{h}}_{n_0} \tag{1.33}$$

exactly the same way as equation (1.28). The MMSE block diagram in Figure 1.12 is the same as the ZF block diagram in Figure 1.11 aside from the noise variance going into the auto-corr block.

CMA Equalizer

The constant modulus algorithm (CMA) equalizer was explored in section **??**. Equation (**??**) updates the CMA filter coefficients using Equation **??** to compute ∇J_{CMA}. The equations are repeated here for convenience with a few changes.

$$\nabla J_{\text{CMA}} \approx \frac{2}{2N_b} \sum_{n=0}^{2N_b-1} \Big[y(n)y^*(n) - R_2\Big]\, y(n)\mathbf{r}^*(n). \tag{1.34}$$

$$\mathbf{c}_{b+1} = \mathbf{c}_b - \mu\nabla J_{\text{CMA}} \tag{1.35}$$

In section **??** equation (1.36) was derived generally, but here we now have known data and equalizer lengths. There is only one $\nabla\mathbf{J}_{\text{CMA}}$ so the subscript CMA is dropped. The adjustment $\nabla\mathbf{J}$ is the same length L_{eq} as the equalizer $\mathbf{c}_n$. The portion of $\mathbf{r}^*(k)$ starts at the center tap of the equalizer L_1. The block index does not need to be tracked because of the frame synconization step, so the subscript $b+1$ and b is replaced with the iteration index n and $n+1$. Because the index n is being used as the iteration index, the summation index is replaced with k. Applying all of these changes results in

$$\nabla\mathbf{J} \approx \sum_{k=0}^{L_{pkt}-1} z(k)\mathbf{r}^*(k). \tag{1.36}$$

$$\mathbf{c}_{n+1} = \mathbf{c}_n - \mu\nabla\mathbf{J} \tag{1.37}$$

where

$$z(k) = \Big[y(k)y^*(k) - R_2 \Big] \, y(k). \tag{1.38}$$

The computation of $\nabla \mathbf{J}$ can be done directly by launching L_{eq} threads to compute a length L_{pkt} summation per thread. But a long summation inside a GPU thread is terribly inefficient. The equation for $\nabla \mathbf{J}$ can be reformulated or massaged to look like a convolutional sum. This will be shown by by doing a toy example.

Suppose that the vectors $\mathbf{r}^*$ and $\mathbf{z}$ are length 4 and zero outside of $0 \leq k \leq 3$. The desired vector $\nabla \mathbf{J}$ is 3 long and L_1 is 1. The toy example computation of $\nabla \mathbf{J}$ is

$$\begin{aligned} \nabla J(0) &= z(0)r(1) + z(1)r(2) + z(2)r(3) \\ \nabla J(1) &= z(0)r(0) + z(1)r(1) + z(2)r(2) + z(3)r(3) \\ \nabla J(2) &= \qquad\qquad z(1)r(0) + z(2)r(1) + z(3)r(2). \end{aligned} \tag{1.39}$$

A convolutional sum of the vector $\mathbf{z}$ with $\mathbf{r}$ is

$$\begin{aligned} y(0) &= z(0)r(0) \\ y(1) &= z(0)r(1) + z(1)r(0) \\ y(2) &= z(0)r(2) + z(1)r(1) + z(2)r(0) \\ y(3) &= z(0)r(3) + z(1)r(2) + z(2)r(1) + z(3)r(0) \\ y(4) &= \qquad\qquad z(1)r(3) + z(2)r(2) + z(3)r(1) \\ y(5) &= \qquad\qquad\qquad\qquad z(2)r(3) + z(3)r(2) \\ y(6) &= \qquad\qquad\qquad\qquad\qquad\qquad z(3)r(3) \end{aligned} \tag{1.40}$$

The convolutional sum looks kind of like the $\nabla \mathbf{J}$ but longer. The desired section of $\mathbf{y}$ is $2 \leq k \leq 4$.

$$\begin{aligned} y(2) &= z(0)r(2) + z(1)r(1) + z(2)r(0) \\ y(3) &= z(0)r(3) + z(1)r(2) + z(2)r(1) + z(3)r(0) \end{aligned} \tag{1.41}$$

$$y(4) = \qquad\qquad z(1)r(3) + z(2)r(2) + z(3)r(1)$$

The set of convolutional equations are structured like $\nabla \mathbf{J}$ but the indices in $r(n)$ are ascending in Equations (1.40) but descending in Equations (1.42). One of the input vectors $\mathbf{z}$ or $\mathbf{r}$ need to be reversed.

Now the indicies are moving in the right directions but the convolutional sum has the wrong structure. The vector $\mathbf{y}$ needs to be reversed also.

$$\begin{aligned} y(4) &= z(0)r(1) + z(1)r(2) + z(2)r(3) \\ y(3) &= z(0)r(0) + z(1)r(1) + z(2)r(2) + z(3)r(3) \\ y(2) &= \qquad\qquad z(1)r(0) + z(2)r(1) + z(3)r(2). \end{aligned} \tag{1.42}$$

Now the convolutional sum $\mathbf{y}$ has been massaged to look just like $\nabla\mathbf{J}$.

$$\begin{aligned} \nabla J(0) &= z(0)r(1) + z(1)r(2) + z(2)r(3) \\ \nabla J(1) &= z(0)r(0) + z(1)r(1) + z(2)r(2) + z(3)r(3) \\ \nabla J(2) &= \qquad\qquad z(1)r(0) + z(2)r(1) + z(3)r(2). \end{aligned} \tag{1.43}$$

This Toy example shows that a simple convolution can be used to calculate $\nabla\mathbf{J}$ instead of a long summation for each index of the gradient. Figure 1.13 shows how to use a convolution to compute $\nabla\mathbf{J}$. One of the input vectors and the output vector must be reversed to compute $\nabla\mathbf{J}$ using a convolution. The L_{eq} long portion of the convolution starts at the index $L_{pkt} - L_2 - 1$. The block diagram in Figure 1.14 shows how the GPU implements the CMA algorithm. First the past filter $\mathbf{c}_n$ is applied by a convolution. The GPU computes $\mathbf{z}$ using L_{pkt} threads per packet, then reverses the vector in preparation for the convolution. The convolution is done as shown in Figure **??**. The GPU then reverses the output vector to obtain $\nabla\mathbf{J}$ from the convolution. With $\nabla\mathbf{J}$ computed, the next CMA filter coefficients $\mathbf{c}_{n+1}$ are updated using L_{eq} threads per packet.

Frequency Domain Equalizer 1

The Frequency Domain Equalizer 1 (FDE1) was studied in section.... Equation .... is the final form of the equalizer. This equation is repeated here

$$\text{FDE}_1 = \frac{\mathbf{H}^*}{\|\mathbf{H}\|^2 + \sigma_0^2} \tag{1.44}$$

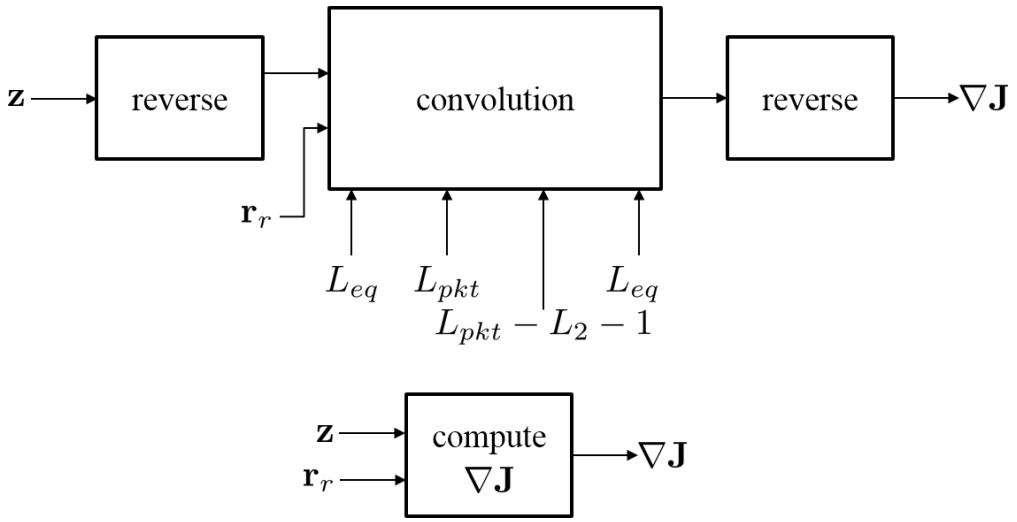

Figure 1.13: The computation of $\nabla\mathbf{J}$ can be done with a convolution of directly.

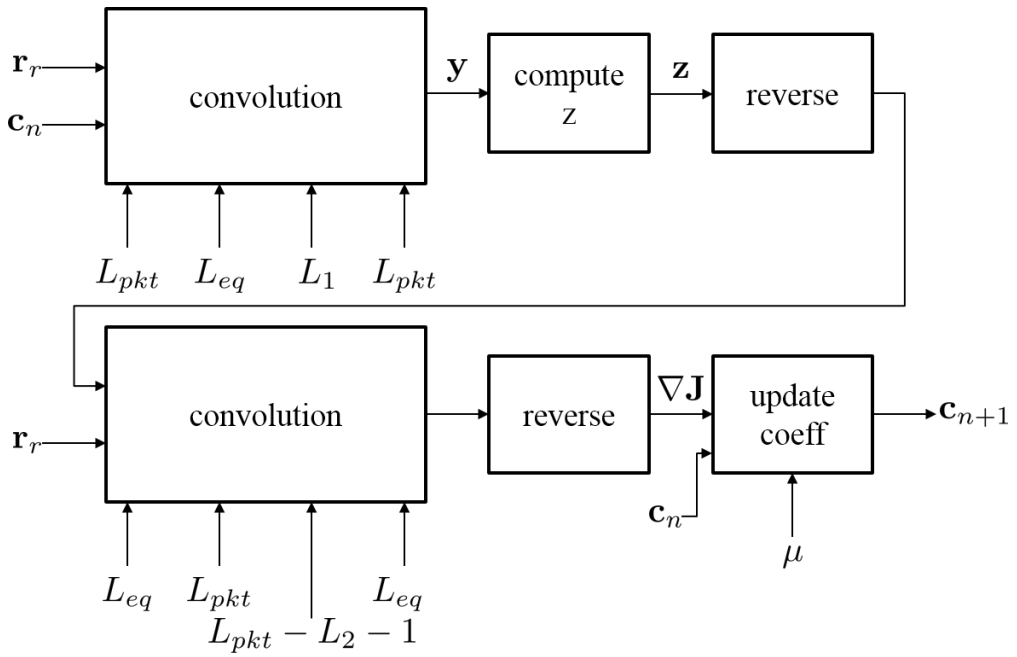

Figure 1.14: The block diagram showing how the GPU implements the CMA algorithm.

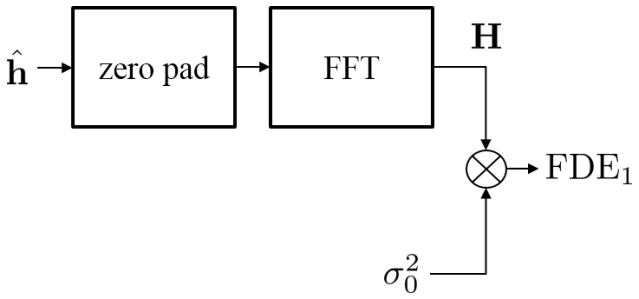

Figure 1.15: The block diagram showing how the GPU calculates the FDE_1 equalizer.

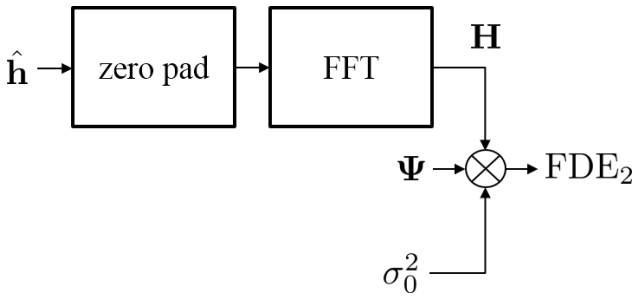

Figure 1.16: The block diagram showing how the GPU calculates the FDE_2 equalizer.

Equation (3.26) is extremely easy to implement in the GPU. All that is need to calculate the equalizer is the Fourier Transform of $\hat{\mathbf{h}}$ and the noise variance σ_0^2. As shown in Figure 1.15, the GPU zero pads that channel estimate then takes the N_{FFT} point FFT to obtain $\mathbf{H}$. Equation (3.26) is then calculated by launching N_{FFT} threads per packet. The output of the multiply FDE_1 is the Frequency Domain Equalizer left in the frequency domain.

Frequency Domain Equalizer 2

The Frequency Domain Equalizer 2 (FDE2) was studied in section.... Equation .... is the final form of the equalizer. FDE_2 is very simmilar to FDE_1, FDE_2 has an extra term on σ_0^2 This equation is repeated here

$$\mathrm{FDE}_2 = \frac{\mathbf{H}^*}{\|\mathbf{H}\|^2 + \sigma_0^2 \mathbf{\Psi}} \tag{1.45}$$

Equation (3.27) is implemented the same way (3.26) but FDE_2 has one extra vector $\mathbf{\Psi}$ in the multiply. The vector $\mathbf{\Psi}$ is recomputed and stored. The output of the multiply FDE_2 is the Frequency Domain Equalizer left in the frequency domain.

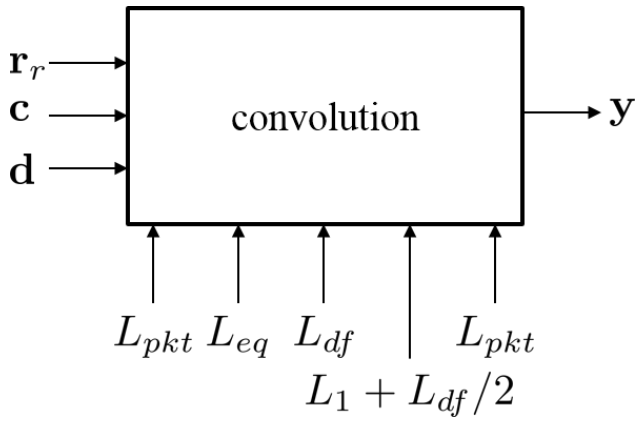

Figure 1.17: The block diagram showing how the GPU applies an FIR equlizer and the Numerically Optimized Perrins detection filter.

1.0.2 Filter Application GPU Implementation

With all the equalizers calculated and ready to apply, the GPU applys the FIR equalizers and the Numerically Optimized detection filter [**?**]. The Perrins detection filter length L_{df} is 21 samples long with the center tap at index 10.

FIR Equalizer Application

The FIR equalizers (ZF, MMSE and CMA) and the Perrins detection filter are all applied using the convolution block shown in Figure **??** and **??**. The derotated samples vector $\mathbf{r}_r$ is L_{pkt} or 12672 samples long. The calculated equalizer $\mathbf{c}$ is L_{eq} or 186 samples long. The Perrins Numerically Optimized detection filter $\mathbf{d}$ is L_{df} or 21 samples long. The L_{pkt} samples that need to be pruned out start at $L_1 + L_{df}/2$. The vector $\mathbf{y}$ are samples that have been equalized and detected, ready for the symbol detector and Phase Lock Loop.

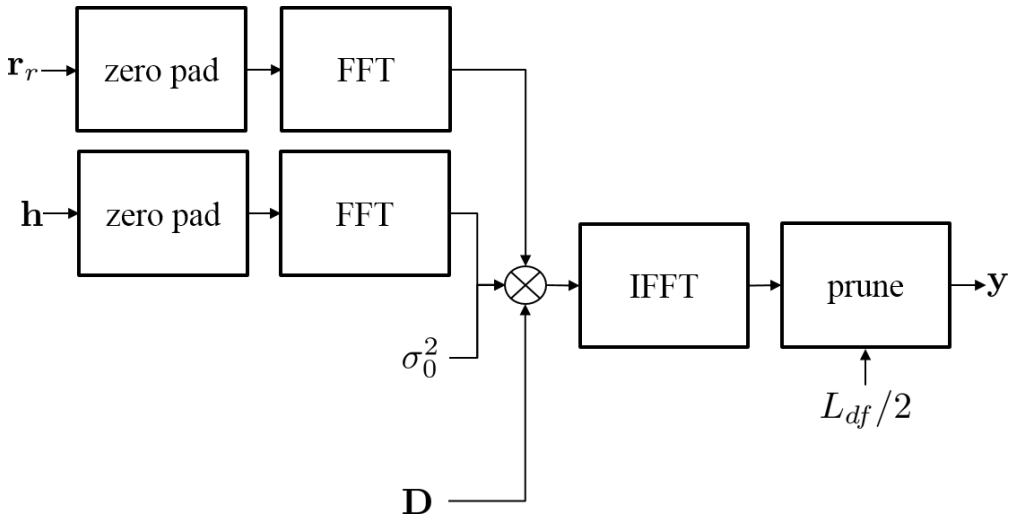

Figure 1.18: The block diagram showing how the GPU applies FDE_1 and the Numerically Optimized Perrins detection filter.

Frequency Domain Equalizer Application

The application of the Frequency Domain Equalizers, FDE_1 and FDE_2, are not applied using the conventional convolution. FDE_1 and FDE_2 are applied in the frequency domain by

$$\text{FDE}_2 = \frac{\mathbf{H}^* \mathbf{R}_r \mathbf{D}}{\|\mathbf{H}\|^2 + \sigma_0^2} \tag{1.46}$$

and

$$\text{FDE}_2 = \frac{\mathbf{H}^* \mathbf{R}_r \mathbf{D}}{\|\mathbf{H}\|^2 + \sigma_0^2 \mathbf{\Psi}} \tag{1.47}$$

where $\mathbf{R}_r$ is the Fourier transform of the zero padded derotated samples $\mathbf{r}_r$ and $\mathbf{D}$ is the Fourier transform of the zero padded detection filter $\mathbf{d}$. The Fourier transform of Perrins Detection filter was initialized and stored.

Figures 1.18 and 1.19 show block diagrams of how FDE_1 and FDE_2 is implemented in the GPU. The vector $\mathbf{\Psi}$ was precomputed and stored at initialization. The multiply in each figure is not a typical multiplication, the multiply block implements Equations (3.26) and (3.27). To prepare the data for the OQPSK demodulator and Phase Lock Loop, the prune block shown in both figures down samples by 2. With the signal downsampled to 1 sample per bit, the equalized received signal is ready for the demodulator, Phase Lock Loop and bit decisions.

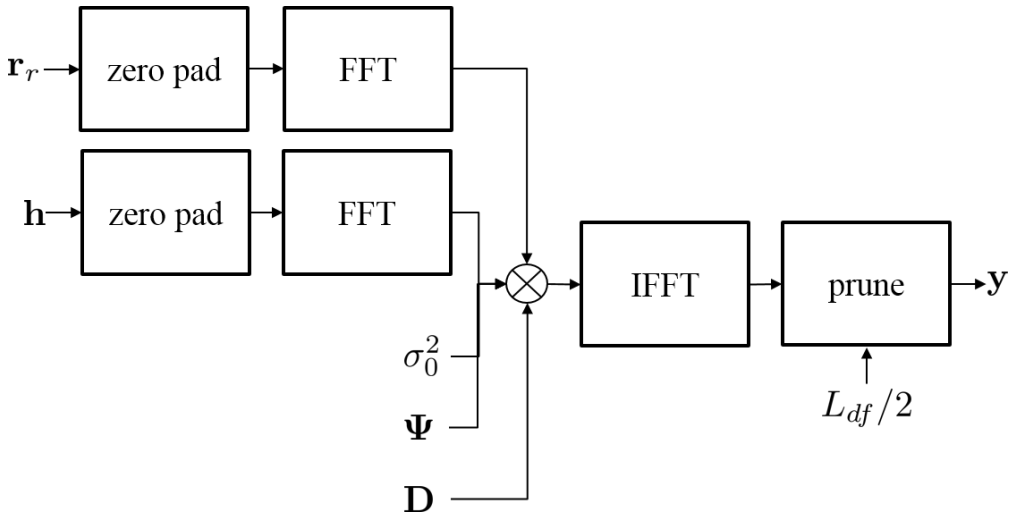

Figure 1.19: The block diagram showing how the GPU applies FDE_2 with $\mathbf{\Psi}$ and the Numerically Optimized Perrins detection filter.

1.0.3 GPU OQPSK Demodulator Implementation

After all the equalizers and detection filters have been applied to the received samples there is 5 different streams of equalized samples. The bit decisions from each stream of equalized samples needs to be obtained.

The top of Figure 1.20 shows a block diagram of how the bit decisions are made. A "symbol-by-symbol" OQPSK detector is used to build the data decision vector $\hat{\mathbf{a}}$ for each stream of equalized samples. The OQPSK detector is first data-aided then decision directed. In case the frequency offset estimator in section 1 was imperfect, a first order with a Phase Lock Loop (PLL) is applied to track out the residual frequency offset. Figure 1.21 shows a block diagram of a first order PLL.

Traditionally a single PLL would be applied to the whole stream of equalized samples because PLLs are inherently serial, but the equalized samples have a packet structure with known data. A PLL can be applied to each packet introducing parallelism to map best to GPUs. One thread per packet per stream of equalized samples is launched to run the OQPSK demodulator shown in Figure 1.21.

Starting at the in the preamble of each packet, the PLL tracks out the a frequency offset for $0 \leq k \leq L_{pkt}$ by estimating the maximum likelihood phase error $e(k)$ where

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

With the estimated phase error estimated, the gain K_1 is applied to $e(k)$ and a Direct Digital Synthesizer (DDS) generates a signal to derotate the equalized sample at index k. For every $y(k)$ in the packet, a derotated sample $y_r(k)$ is calculated and the data decision $\hat{a}(k)$ is estimated where

$$\hat{a}(k) = \begin{cases} p(k) & k < L_p + L_{asm} \\ sgn(\mathbb{R}\{y_r(k)\}) & k \geq L_p + L_{asm} \quad \& \quad k \text{ even} \\ sgn(\mathbb{I}\{y_r(k)\}) & k \geq L_p + L_{asm} \quad \& \quad k \text{ odd} \end{cases}. \tag{1.48}$$

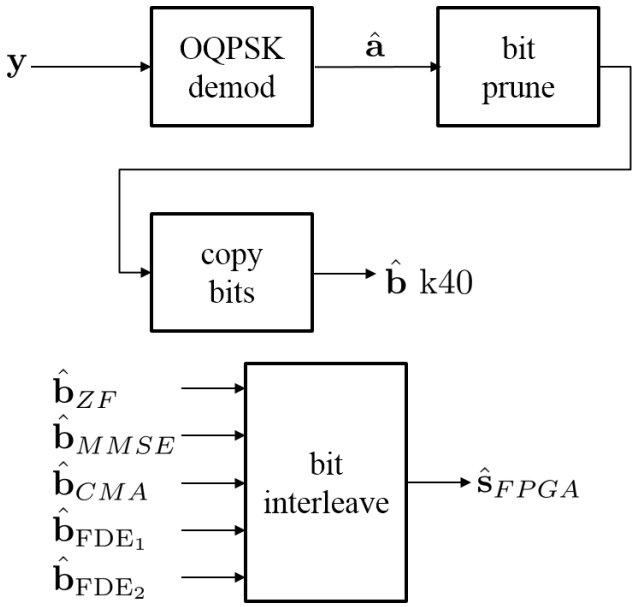

Figure 1.20: The block diagram showing how the GPU applies the demodulator to equalized received samples to obtain the bit decisions.

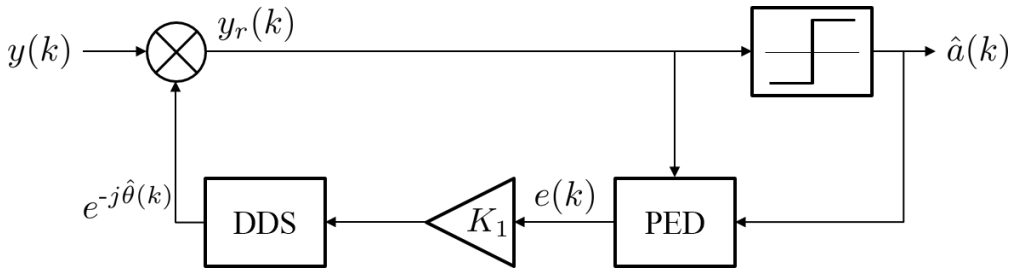

Figure 1.21: The block diagram showing the OQPSK demodulator from Figure 1.20. The $\hat{a}(k)$ is data-aided for $k < L_p + L_{asm}$ and decision directed when $k \geq L_p + L_{asm}$.

and $p(k)$ are the known bits or decisions in the preamble and ASM. Once the index k is out of the preamble and asm, the data decisions $\hat{a}(k)$ are based on the sign of the real $\mathbb{R}$ or imaginary $\mathbb{I}$part of $y_r(k)$.

After the vector $\hat{\mathbf{a}}$ is calculated, the data bits are pruned out of every packet as shown in Figure 1.20 as the bit prune block. One thread per data bit is launched to strip the data bits and build the bit decision vector $\hat{\mathbf{b}}$ from $\hat{\mathbf{a}}$ where

$$\hat{b}(n) = \begin{cases} 0 & \hat{a}(n) > 0 \\ 1 & \hat{a}(n) < 0 \end{cases}. \tag{1.49}$$

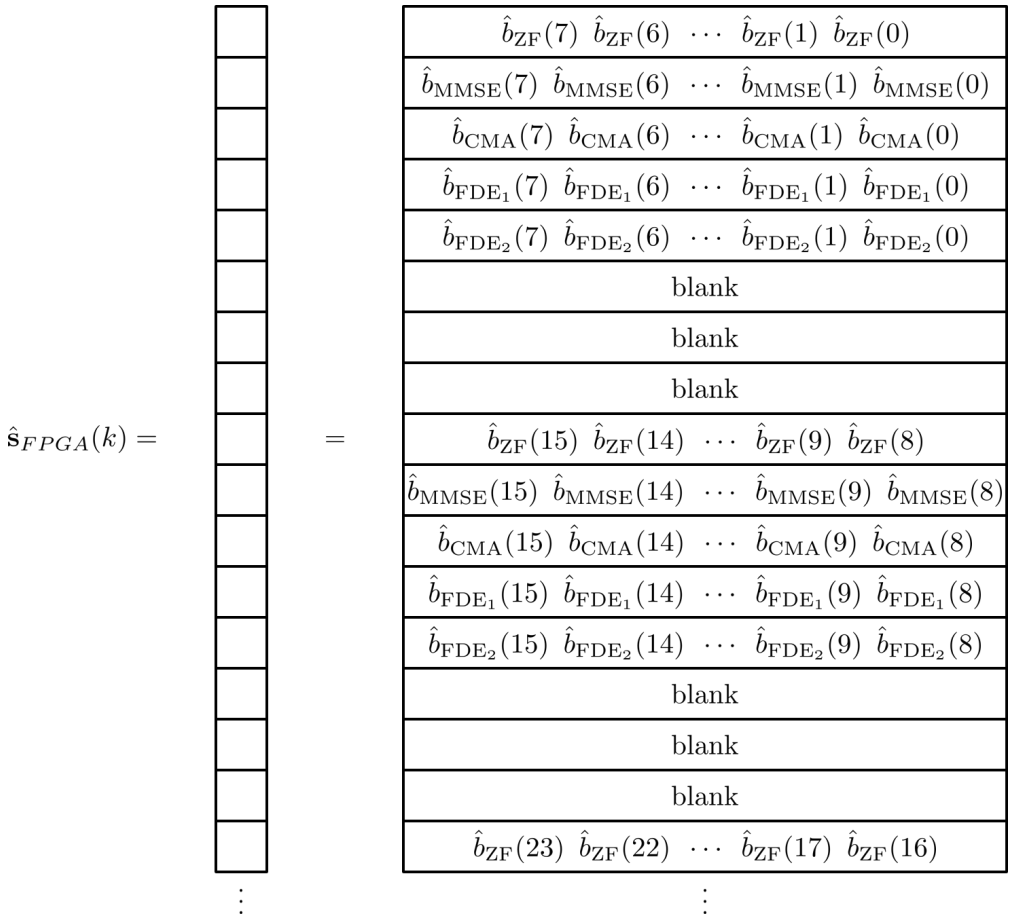

Figure 1.22: The bit stream vector $\hat{\mathbf{s}}_{FPGA}$ is array chars. The bit streams are interleaved and each char contains 8 bit decisions from a single bit stream.

After the data bit vector $\hat{\mathbf{b}}$ for each equalized stream is built, the bit decisions are all transferred to the Tesla k40 GPU in the copy bits block. The bit vector $\hat{\mathbf{s}}_{FPGA}$ is a c++ char array containing interleaved bit streams $\hat{\mathbf{b}}$ from each stream of equalized samples. In each char index of $\hat{\mathbf{s}}_{FPGA}$ contains 8 bit decisions for a single equalized bit stream. The first bit stream in $\hat{\mathbf{s}}_{FPGA}$ is the bit stream from the ZF equalizer, followed by MMSE, CMA, FDE_1, FDE_2 then 3 blank streams. Figure 1.22 shows how the pattern repeats every 8 indices in the char array $\hat{\mathbf{s}}_{FPGA}$. This array $\hat{\mathbf{s}}_{FPGA}$ containing bit decisions from each equalizer is burst into the FPGA then clocked out to the Bit Error Rate Tester.

Chapter 2

System Overview

2.1 Overview

This thesis gives and overview of the the GPU implementation and performance of data aided equalizes. but the focus of this thesis is on the computation and application of the equalizers. Before data-aided equalizers can be computed and applied: the preambles must found, the signal packetized, the signal ”de-rotated” then the channel and noise variance estimated from the de-rotated signal. The equalizers are then computed and applied using the de-rotated signal and the estimated channel and noise variance. After the equalizers have been applied a OQPSK detector is applied to the output of each equalizer. A simple block Diagram is shown in Figure 2.1.

This chapter will proceed as follows, section 2.1.1 will explain the GPU implementation of the finding the preambles and packetizing the received signal, section 2.1.2 will explain the GPU implementation of estimating the frequency offset and de-rotating the signal, section 2.1.3 will explain the GPU implementation of estimating the channel, section 2.1.4 will explain the GPU implementation of estimating the noise variance, section 2.1.5 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 3.

2.1.1 Preamble Detection

To compute preamble assisted equalizers, estimators use the preamble to estimate various parameters. The details of the Preamble Detector is explained in [1]. The iNET packet is comprised of three different sections: preamble, asynchronous marker (ASM) and the data. The packet and received sample structure is shown in Figure2.2.

The goal of the Preamble Detection step is to structure the received samples into packets or ”packetize” the samples. To packetize, the received samples are put to into vectors with the start of

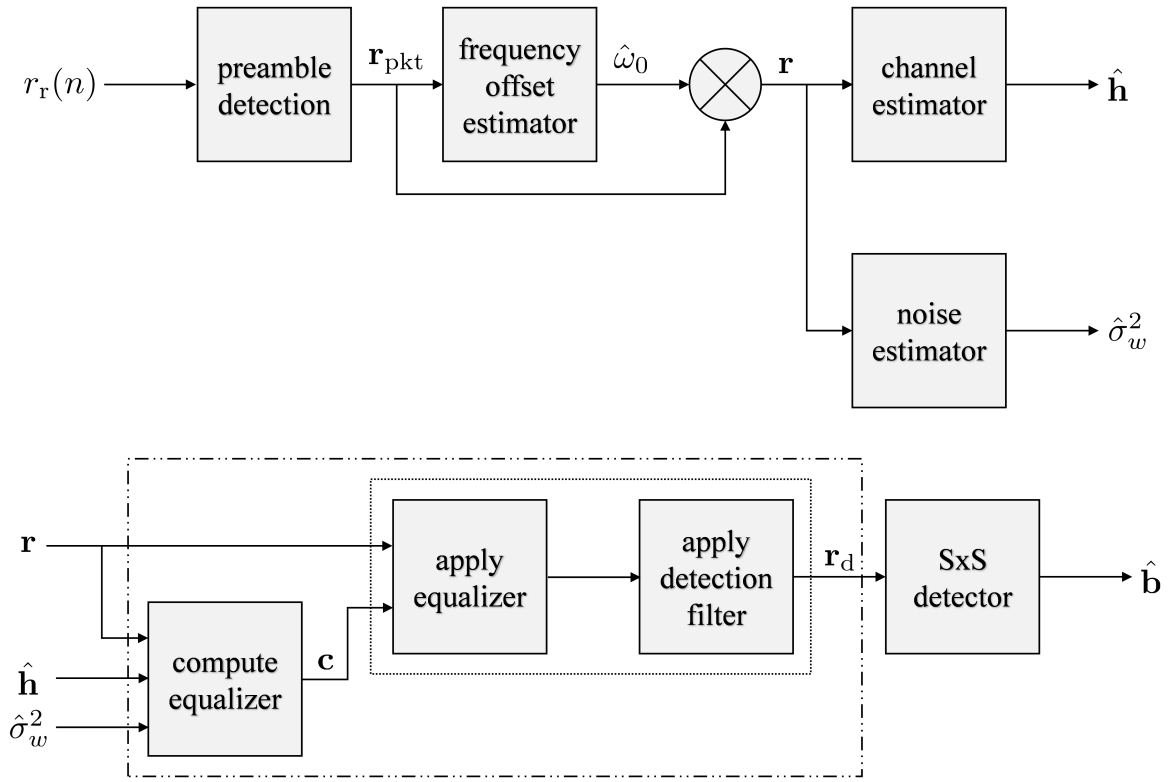

Figure 2.1: This a simple block diagram of what the GPU does.

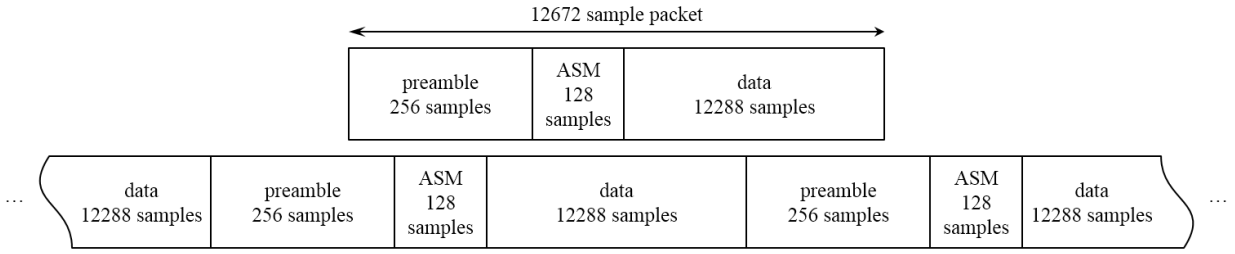

Figure 2.2: The iNET packet structure.

each preamble as the first element. To summarize the frame synchronization step, the preambles in the received samples are found using a preamble detector. The preamble detector output is seaching using algorithms to scan the output and find the starting index of each packet. Using the starting index of each packet, the received samples are packetized into vectors.

To find the preambles in the batch, the preamble detector computs the sample correlation function between the received samples and a stored local copy of the known samples of the iNET preamble. A search algorithm then search the correlation function for sample indices that correlate

strongest to the preamble. A spike in the correlation function indicates the start of a preamble in the received samples. Finally, using the starting indices of each packet, received samples are structured and synchronized into frames or packets.

The first step in the frame synchronizer is to compute the sample correlation between the received samples and the preamble. A lower complexity preamble detector is shown in equation (**??**)-(**??**) in section **??** and repeated here for convenience.

$$L(n) = \sum_{m=0}^{7} \left[I^2(n,m) + Q^2(n,m) \right] \tag{2.1}$$

where

$$\begin{aligned} I(n,m) \approx & \sum_{\ell \in \mathcal{L}_1} r_R(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_2} r_R(\ell+32m+n) + \sum_{\ell \in \mathcal{L}_3} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_4} r_I(\ell+32m+n) \\ & + 0.7071 \left[\sum_{\ell \in \mathcal{L}_5} r_R(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_6} r_R(\ell+32m+n) \right. \\ & \left. + \sum_{\ell \in \mathcal{L}_7} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_8} r_I(\ell+32m+n) \right], \end{aligned} \tag{2.2}$$

$$\begin{aligned} Q(n,m) \approx & \sum_{\ell \in \mathcal{L}_1} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_2} r_I(\ell+32m+n) \\ & - \sum_{\ell \in \mathcal{L}_3} r_R(\ell+32m+n) + \sum_{\ell \in \mathcal{L}_4} r_R(\ell+32m+n) \\ & + 0.7071 \left[\sum_{\ell \in \mathcal{L}_5} r_I(\ell+32m+n) - \sum_{\ell \in \mathcal{L}_6} r_I(\ell+32m+n) \right. \\ & \left. - \sum_{\ell \in \mathcal{L}_7} r_R(\ell+32m+n) + \sum_{\ell \in \mathcal{L}_8} r_R(\ell+32m+n) \right] \end{aligned} \tag{2.3}$$

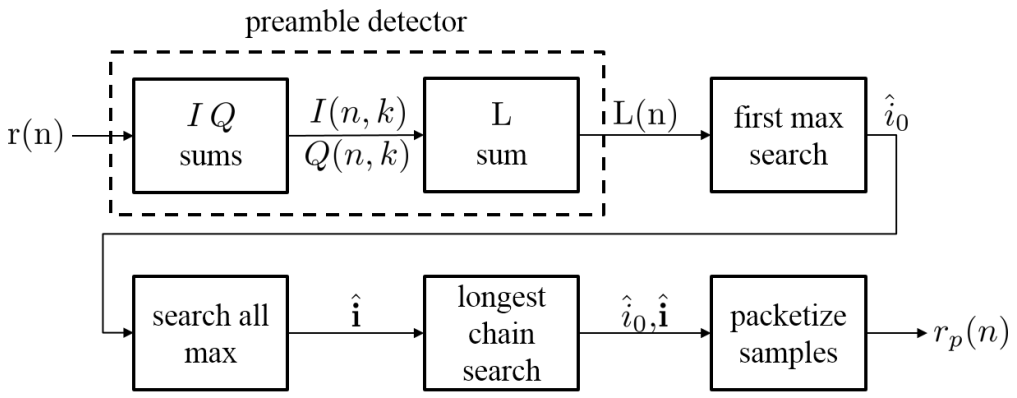

Figure 2.3: The block diagram for the frame synchronization implementation.

with

$$\begin{aligned}
\mathcal{L}_1 &= \{0, 8, 16, 24\} \\
\mathcal{L}_2 &= \{4, 20\} \\
\mathcal{L}_3 &= \{2, 10, 14, 22\} \\
\mathcal{L}_4 &= \{6, 18, 26, 30\} \\
\mathcal{L}_5 &= \{1, 7, 9, 15, 17, 23, 25, 31\} \\
\mathcal{L}_6 &= \{3, 5, 11, 12, 13, 19, 21, 27, 28, 29\} \\
\mathcal{L}_7 &= \{1, 3, 9, 11, 12, 13, 15, 21, 23\} \\
\mathcal{L}_8 &= \{5, 7, 17, 19, 25, 27, 28, 29, 31\}.
\end{aligned} \tag{2.4}$$

The preamble detector is implemented in the GPU in two kernels as shown by the dotted box in Figure 2.3. The first kernel computes the inner summations and the second computes the outer summation.

The inner summation, as defined in equations (2.2)-(2.4), computes an I and Q pair. A single I and Q pair is computed by summing 32 scaled real or imaginary parts of received samples. For each received sample an I and Q pair is computed.

The outer summation, as defined in equation (2.1), computes the reduced complexity maximum likelihood correlation L. One sample of L is computed by summing 8 squared then summed I and Q pairs. For each received sample L is computed.

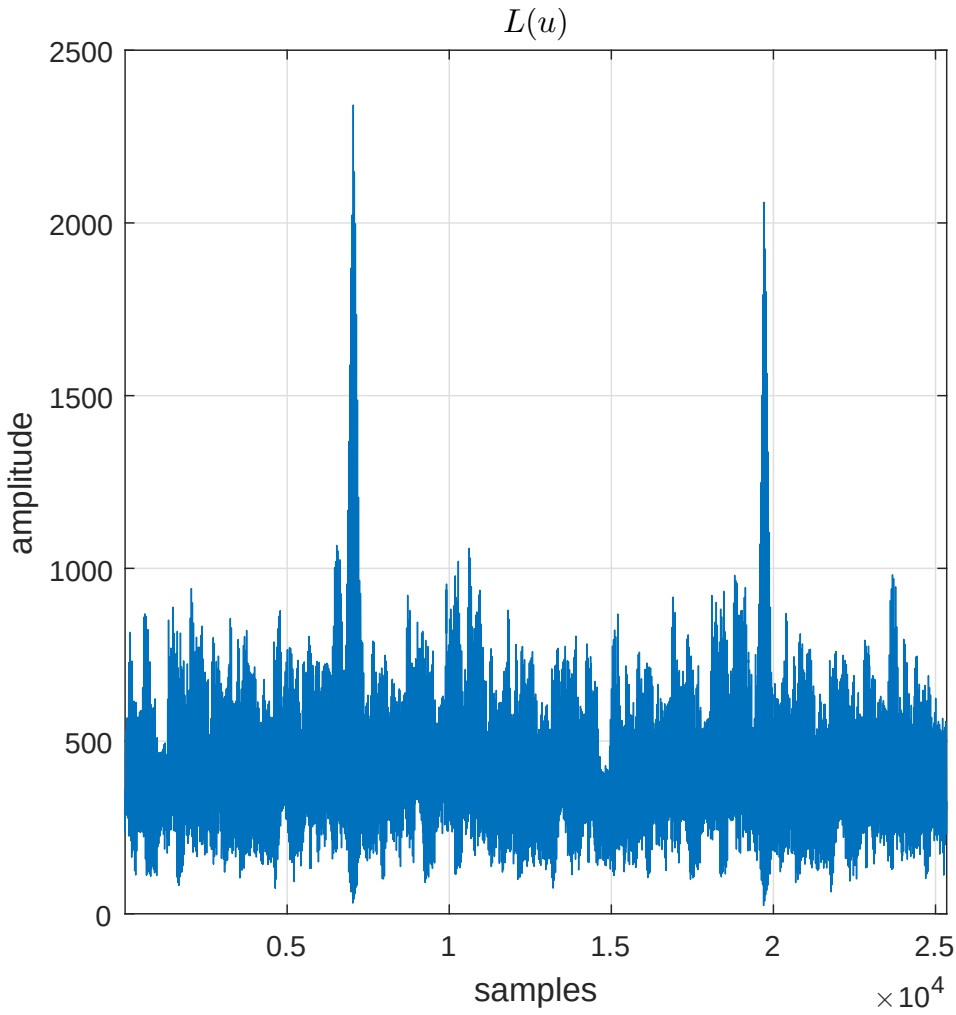

Figure 2.4: The output of the Preamble Detector $L(u)$.

Figure 2.4 shows an example of the first $2 \times L_{pkt}$ samples of L. The local maximums in L in the figure indicate a preamble starts at the maximum's sample index. The first local maximum is at sample index 7040, indicating the first packet starts at that index. The figure also shows the second packet starts at sample index 19712. The difference between these sample indices is L_{pkt}, this difference agrees with the packet length shown in Figure 2.2.

Because of the structure of the preamble, the preamble detector output L has some unique properties. The Figures in 2.5 show the correlation function around an expected preamble location. The correlation functions have peaks every 32 samples because the preamble bit sequence comprises eight repetitions of the 16-bit pattern CD98_{hex}. The repetitive structure causes one main correlation peak with seven side peaks.

When ideal samples are received the preamble detector output looks like Figure 2.5(a). The structure of the correlation peaks still occur when the signal to noise ratio is low, as shown in Figure 2.5(b). But when the signal to noise ratio is low and major multipath distortion happen, the correlation peaks look like Figures 2.5(c) and (d). The structure of the correlation peaks can cause a simple algorithm to find an incorrect preamble starting location.

In the worst case scenario, a simple algorithm might find an incorrect preamble index by searching a poorly placed search window. A poorly placed window might search the large side correlation peaks from Figure 2.5(a) and the small main correlation peaks from Figures 2.5(c) or (d). Because the first three side peaks in Figure 2.5(a) are much taller than the main peak in Figures 2.5(c) and (d), an incorrect preamble starting indices will result if search windows are not defined safely.

To prevent searching multiple preamble correlation peaks, search windows should only search the correlation peak of one preamble. To ensure the correlation peaks from only one preamble is searched, windows of length L_{pkt} are centered on expected preamble starting locations. Figure 2.6 shows an example of safe search windows centered on expected preamble starting locations.

To define safe search windows, a rough estimate of the preamble starting locations is made. The GPU launches a kernel with one thread to search for the maximum in the first L_{pkt} samples of the received samples. The maximum index is saved as $\hat{i}_0$. $\hat{i}_0$ is used to make a rough estimate for all the preamble staring locations by building a vector defined as

$$\hat{\mathbf{i}} = \begin{bmatrix} \hat{i}_0 + 0 \times L_{pkt} \\ \hat{i}_0 + 1 \times L_{pkt} \\ \vdots \\ \hat{i}_0 + 3102 \times L_{pkt} \\ \hat{i}_0 + 3103 \times L_{pkt} \end{bmatrix} \tag{2.5}$$

With rough estimates of where the preambles should be located, the GPU searches safe windows of L for local maximums. On local maximum is found in each search window by centering each window on elements from $\hat{\mathbf{i}}$. The GPU launches one thread per search window to find the

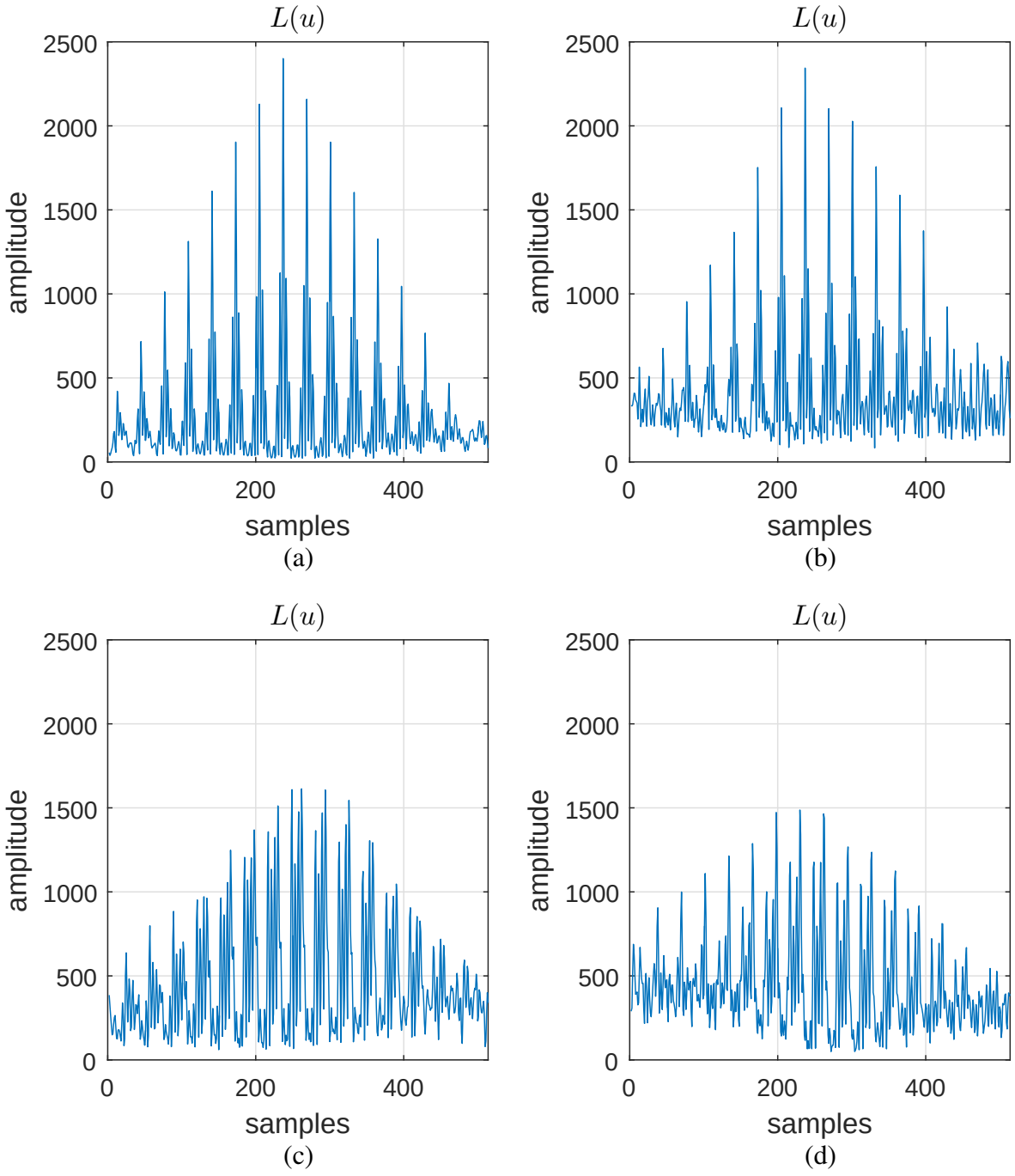

Figure 2.5: Detailed view of $L(u)$. (a): correlation peaks of a distortion free and noiseless signal; (b): correlation peaks of a distortion free but noisy signal with $E_b/N_0 = 0$dB; (c): correlation peaks of a distorted and noisy signal with $E_b/N_0 = 0$dB; (d): correlation peaks of a distorted and noisy signal with $E_b/N_0 = 0$dB

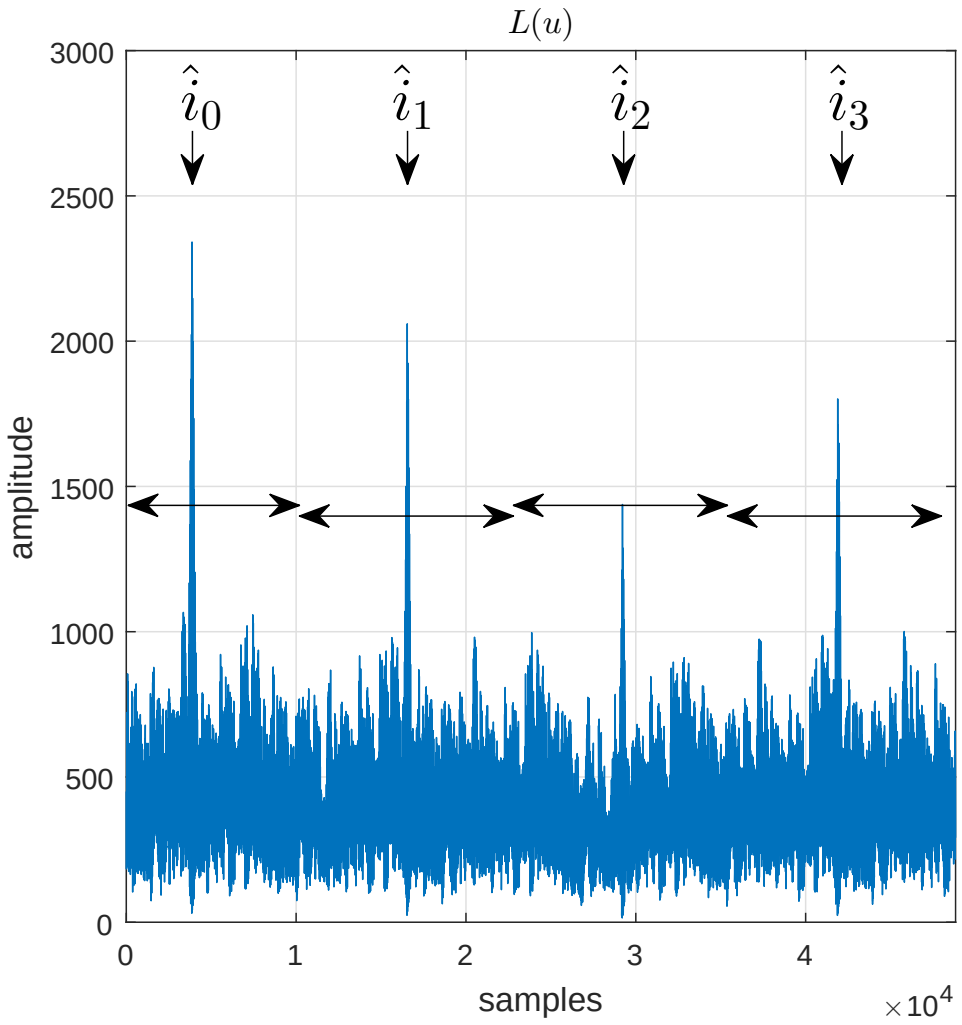

Figure 2.6: Safe search windows defined to search only one preamble correlation peak.

local maximum and save its sample index in $\hat{\mathbf{i}}$. The vector $\hat{\mathbf{i}}$ has the sample indices for 3103 local maximums that should be L_{pkt} samples apart.

Because of noise and multipath, the sample indices in $\hat{\mathbf{i}}$ are not the ideal L_{pkt} samples. The GPU launches one thread to search $\hat{\mathbf{i}}$ for the longest chain of perfectly spaced indices. The modulo of the last index in the longest chain of indices spaced L_{pkt} samples apart is the best estimate for $\hat{i}_0$. Once again, $\hat{i}_0$ is updated with the best estimated first preamble starting location. The vector $\hat{\mathbf{i}}$ is also updated again as in equation (2.5).

Now with the best estimate of the preamble starting locations, the received samples can be packetized and synchronized. Figure 2.7 shows the relationship of $\hat{\mathbf{i}}$ and $\boldsymbol{r}$. Using $\hat{\mathbf{i}}$, the GPU launches one thread per received sample to packetize $\boldsymbol{r}$ into $\boldsymbol{r}_p$ as shown in Figure 2.8. The

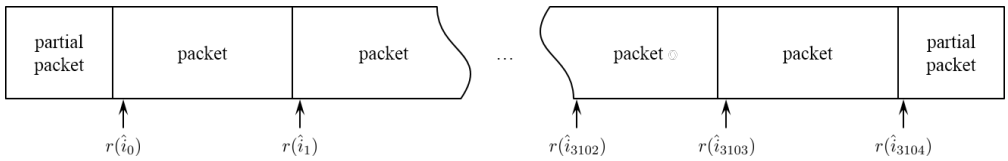

Figure 2.7: The starting sample index for each packet in the batch.

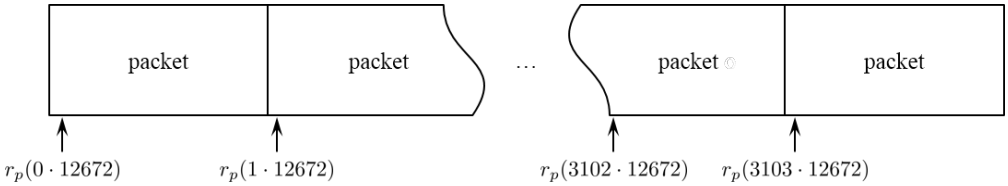

Figure 2.8: The packetized structure of the received signals after the frame synchronization step.

structure of r_p majorly simplifies indexing and removes the need for $\hat{\mathbf{i}}$.

$$\boldsymbol{r}_p = \begin{bmatrix} r(\hat{i}_0) \\ r(\hat{i}_0 + 1) \\ \vdots \\ r(\hat{i}_0 + 12670) \\ r(\hat{i}_0 + 12671) \\ r(\hat{i}_1) \\ r(\hat{i}_1 + 1) \\ \vdots \\ r(\hat{i}_{3103} + 12670) \\ r(\hat{i}_{3103} + 12671) \end{bmatrix} \tag{2.6}$$

The steps in Figure 2.3 synchronize the received samples r into r_p by using a preamble detector. The output of the preamble detector, L, is searched for local maximums. Imperfect spacing of local maximums is fixed by finding the longest chain of perfectly spaced indices in $\hat{\mathbf{i}}$. The received samples are then packetized based on the corrected preamble spacing.

2.1.2 Frequency Offset Estimation and Compensation

2.1.3 Channel Estimation

2.1.4 Noise Variance Estimation

2.1.5 OQPSK Detector

Chapter 3

Equalizer Equations

3.1 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

3.2 Equations

3.2.1 The Solving Equalizers

The Zero-Forcing Equalizer

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

$$\mathbf{c}_{\text{ZF}} = (\mathbf{H}^{\dagger}\mathbf{H})^{-1}\mathbf{H}^{\dagger}\mathbf{u}_{n_0} \tag{3.1}$$

where $\mathbf{c}_{\text{ZF}}$ is a $L_{eq} \times 1$ vector of equalizer coefficients computed to invert the channel estimate $\mathbf{h}$. The channel estimate is used to build the $L_{eq} + N_1 + N_2 \times L_{eq}$ convolution matrix

$$\mathbf{H} = \begin{bmatrix} h(-N_1) & & & \\ h(-N_1+1) & h(-N_1) & & \\ \vdots & \vdots & \ddots & \\ h(N_2) & h(N_2-1) & & h(-N_1) \\ & h(N_2) & & h(-N_1+1) \\ & & & \vdots \\ & & & h(N_2) \end{bmatrix}. \tag{3.2}$$

and $\mathbf{u}_{n_0}$ is the desired channel impulse response centered on $n0 = N_1 + L_1 + 1$

$$\mathbf{u}_{n_0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} n_0 - 1 \text{ zeros} \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} N_1 + N_2 + L_1 + L_2 - n_0 + 1 \text{ zeros} \end{matrix} . \tag{3.3}$$

The computation of the coefficients in Equation (3.1) 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

$$\mathbf{R}_h = \mathbf{H}^\dagger\mathbf{H} = \begin{bmatrix} r_h(0) & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) \end{bmatrix} \tag{3.4}$$

where

$$r_h(k) = \sum_{n=-N_1}^{N_2} h(n)h^*(n-k). \tag{3.5}$$

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

$$\mathbf{h}_{n_0} = \mathbf{H}^\dagger\mathbf{u}_{n_0} = \begin{bmatrix} h(L_1) \\ \vdots \\ h(0) \\ \vdots \\ h(-L_2) \end{bmatrix} . \tag{3.6}$$

Replacing the matrix multiplication $\mathbf{H}^\dagger\mathbf{H}$ and $\mathbf{H}^\dagger\mathbf{u}_{n_0}$ simplifies Equation (3.1) to

$$\mathbf{c}_{\text{ZF}} = \mathbf{R}_h^{-1}\mathbf{h}_{n_0}. \tag{3.7}$$

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 $\mathbf{R}_h$ can be leveraged to reduce computation drastically. The Zero-Forcing Equalizer coefficients are computed by solving

$$\mathbf{R}_h\mathbf{c}_{\text{ZF}} = \mathbf{h}_{n_0}. \tag{3.8}$$

MMSE Equalizer

The MMSE equalizer was studied in the PAQ Phase 1 Final Report in equation 330.

$$\mathbf{c}_{MMSE} = \left[\mathbf{G}\mathbf{G}^\dagger + \frac{\sigma_w^2}{\sigma_s^2}\mathbf{I}_{L_1+L_2+1}\right]\mathbf{g}^\dagger \tag{3.9}$$

where

$$\mathbf{G} = \begin{bmatrix} h(N_2) & \cdots & h(-N_1) & & & \\ & h(N_2) & \cdots & h(-N_1) & & \\ & & \ddots & & \ddots & \\ & & & h(N_2) & \cdots & h(-N_1) \end{bmatrix} \tag{3.10}$$

and

$$\mathbf{g} = \begin{bmatrix} h(L_1) \cdots h(-L_2) \end{bmatrix}. \tag{3.11}$$

The matrix multiplication $\mathbf{G}\mathbf{G}^\dagger$ is the same autocorrelation matrix $\mathbf{R}_h$ as Equation (3.4). The vector $\mathbf{g}^\dagger$ is also the same vector as $\mathbf{h}_{n_0}$. The signal-to-noise ratio estimate $\frac{1}{2\hat{\sigma_w^2}}$ is substituted in for the fraction $\frac{\sigma_w^2}{\sigma_s^2}$ using Equation 333 Rice's report. Equation (3.9) can be reformulated to

$$\left[\mathbf{R}_h + \frac{1}{2\hat{\sigma_w^2}}\mathbf{I}_{L_1+L_2+1}\right]\mathbf{c}_{\text{MMSE}} = \mathbf{h}_{n_0}. \tag{3.12}$$

The MMSE Equalizer coefficients are solved for in a similar fashion to Zero-Forcing is in Equation (3.8). The only difference between Equation (3.12) and (3.8) is the noise variance is added down the diagonal of $\mathbf{R}_h$. The matrix $\mathbf{R}_{hw}$ is defined to make the computation of the MMSE Equalizer coefficients the same as the Zero-Forcing Equalizer coefficients by adding the noise variance to the diagonal of $\mathbf{R}_h$

$$\mathbf{R}_{hw} = \mathbf{H}^{\dagger}\mathbf{H} = \begin{bmatrix} r_h(0) + \frac{1}{2\hat{\sigma_w^2}} & r_h^*(1) & \cdots & r_h^*(L_{eq}-1) \\ r_h(1) & r_h(0) + \frac{1}{2\hat{\sigma_w^2}} & \cdots & r_h^*(L_{eq}-2) \\ \vdots & \vdots & \ddots & \\ r_h(L_{eq}-1) & r_h(L_{eq}-2) & \cdots & r_h(0) + \frac{1}{2\hat{\sigma_w^2}} \end{bmatrix}. \tag{3.13}$$

The MMSE Equalizer coefficients are computed by solving

$$\mathbf{R}_{hw}\mathbf{c}_{\text{MMSE}} = \mathbf{h}_{n_0}. \tag{3.14}$$

3.2.2 The Iterative Equalizer

The Constant Modulus Algorithm

CMA uses a steepest decent algorithm.

$$\mathbf{c}_{b+1} = \mathbf{c}_b - \mu \nabla \mathbf{J} \tag{3.15}$$

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

$$\nabla J = \frac{2}{L_{pkt}} \sum_{n=0}^{L_{pkt}-1} \Big[y(n)y^*(n) - R_2 \Big] \, y(n)\mathbf{r}^*(n). \tag{3.16}$$

where

$$\mathbf{r}(n) = \begin{bmatrix} r(n+L_1) \\ \vdots \\ r(n) \\ \vdots \\ r(n-L_2) \end{bmatrix}. \tag{3.17}$$

This means ∇J is of the form

$$\nabla J = \begin{bmatrix} \nabla J(-L_1) \\ \vdots \\ \nabla J(0) \\ \vdots \\ \nabla J(L_2) \end{bmatrix}. \tag{3.18}$$

To Leverage computational efficiency of FFT, re-express the elements of ∇J as a convolution.

To begin define

$$z(n) = 2 \Big[y(n)y^*(n) - R_2 \Big] \, y(n) \tag{3.19}$$

so that ∇J may be expressed as

$$\nabla J = \frac{z(0)}{L_{pkt}} \begin{bmatrix} r^*(L_1) \\ \vdots \\ r^*(0) \\ \vdots \\ r^*(L_2) \end{bmatrix} + \frac{z(1)}{L_{pkt}} \begin{bmatrix} r^*(1+L_1) \\ \vdots \\ r^*(1) \\ \vdots \\ r^*(1-L_2) \end{bmatrix} + \cdots \frac{z(L_{pkt}-1)}{L_{pkt}} \begin{bmatrix} r^*(L_{pkt}-1+L_1) \\ \vdots \\ r^*(L_{pkt}-1) \\ \vdots \\ r^*(L_{pkt}-1-L_2) \end{bmatrix}. \quad (3.20)$$

The kth value of ∇J is

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m) r^*(m-k), \quad -L_1 \leq k \leq L_2. \quad (3.21)$$

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

$$\rho(n) = r^*(n). \quad (3.22)$$

Now

$$\nabla J(k) = \frac{1}{L_{pkt}} \sum_{m=0}^{L_{pkt}-1} z(m) \rho(k-m). \quad (3.23)$$

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

$$\begin{aligned} b(n) &= \sum_{m=0}^{L_{pkt}-1} z(m)\rho(n-m) \\ &= \sum_{m=0}^{L_{pkt}-1} z(mr^*(m-n) \end{aligned} \quad (3.24)$$

Comparing Equation (3.23) and (3.24) shows that

$$\nabla J(k) = \frac{1}{L_{pkt}} b(k), \quad -L_1 \leq k \leq L_2. \quad (3.25)$$

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

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

3.2.3 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

$$C_{\text{FDE1}}(\omega) = \frac{\hat{H}^*(\omega)}{|\hat{H}(\omega)|^2 + \frac{1}{\hat{\sigma}^2}} \tag{3.26}$$

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

The Frequency Domain Equalizer One

The Frequency Domain Equalizer Twe (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

$$C_{\text{FDE2}}(\omega) = \frac{\hat{H}^*(\omega)}{|\hat{H}(\omega)|^2 + \frac{\Psi(\omega)}{\hat{\sigma}^2}} \tag{3.27}$$

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 3.1. FDE2 needs no massaging because Equation (3.27) is easily implemented in the GPU and is computationally efficient.

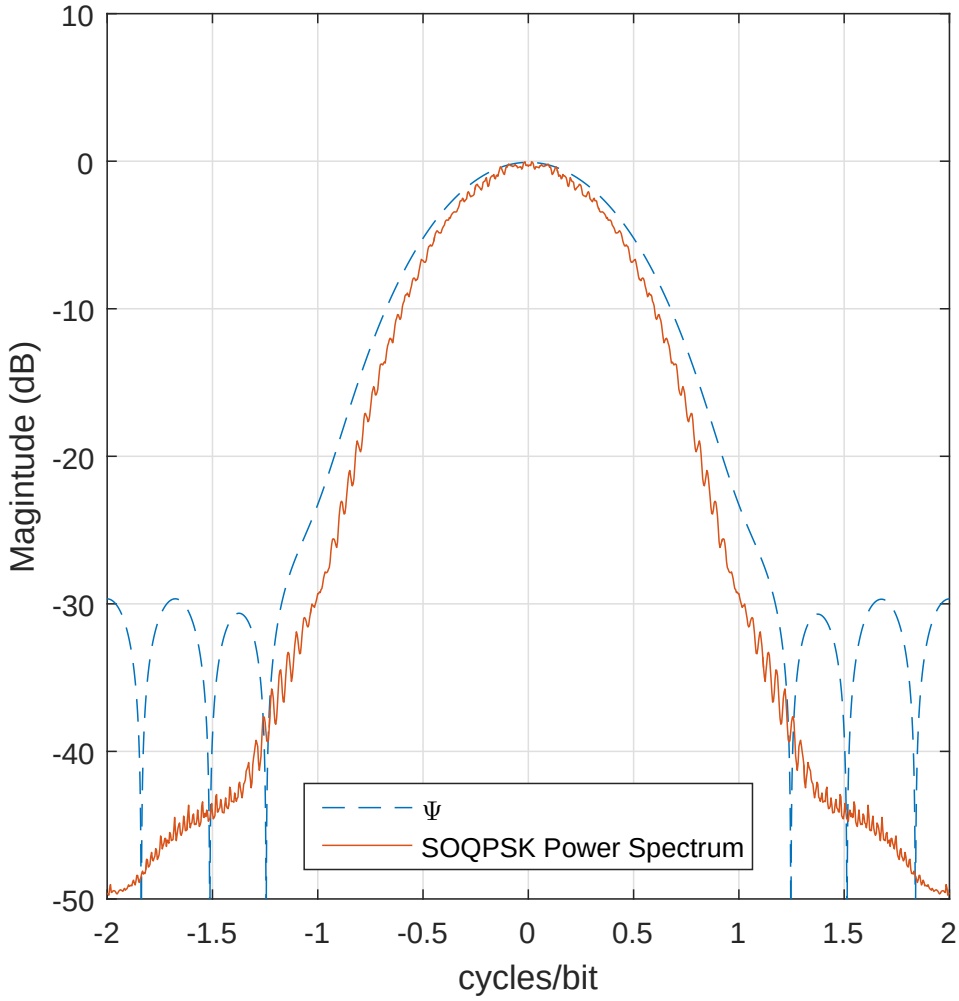

Figure 3.1: A block diagram illustrating organization of the algorithms in the GPU.

Bibliography

[1] M. Rice and A. Mcmurdie, "On frame synchronization in aeronautical telemetry," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 52, no. 5, pp. 2263–2280, October 2016. 31