GPU Specialization Capstone Project

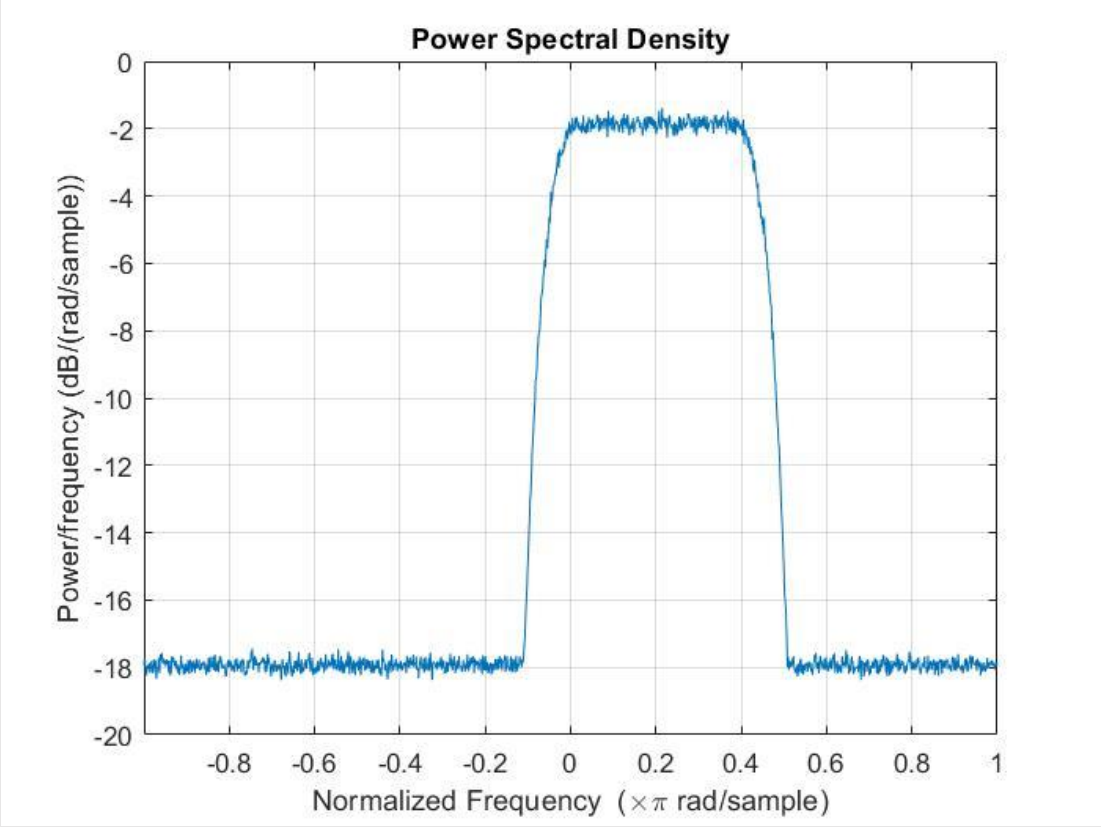
In The Name Of God

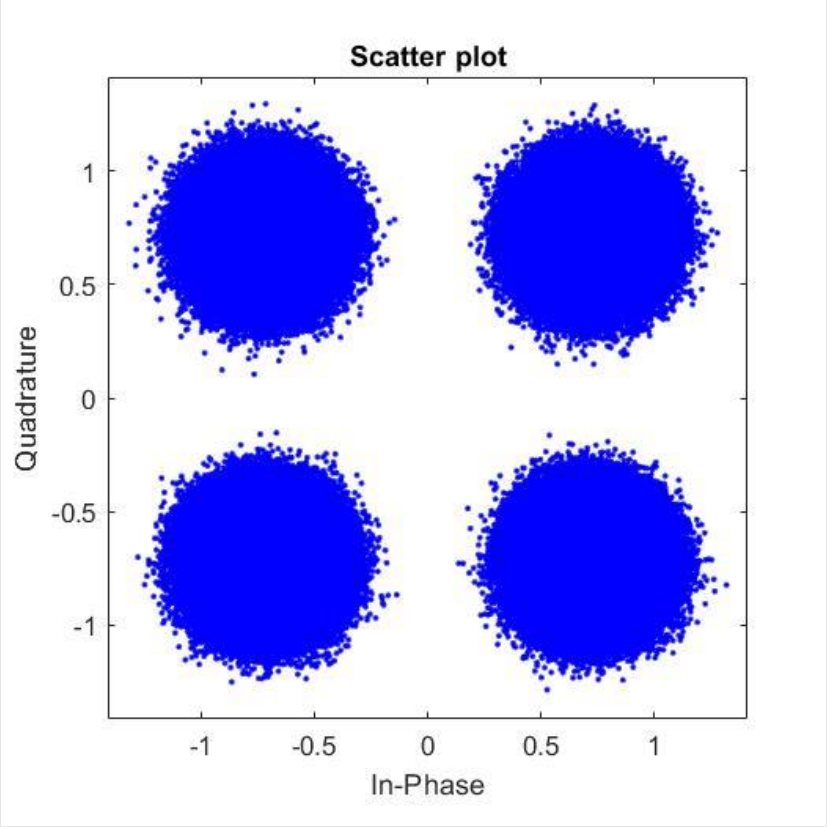
The implementation of Software Defined Radio based on GPU

Created by:
Mohammad Hasan Shammakhi

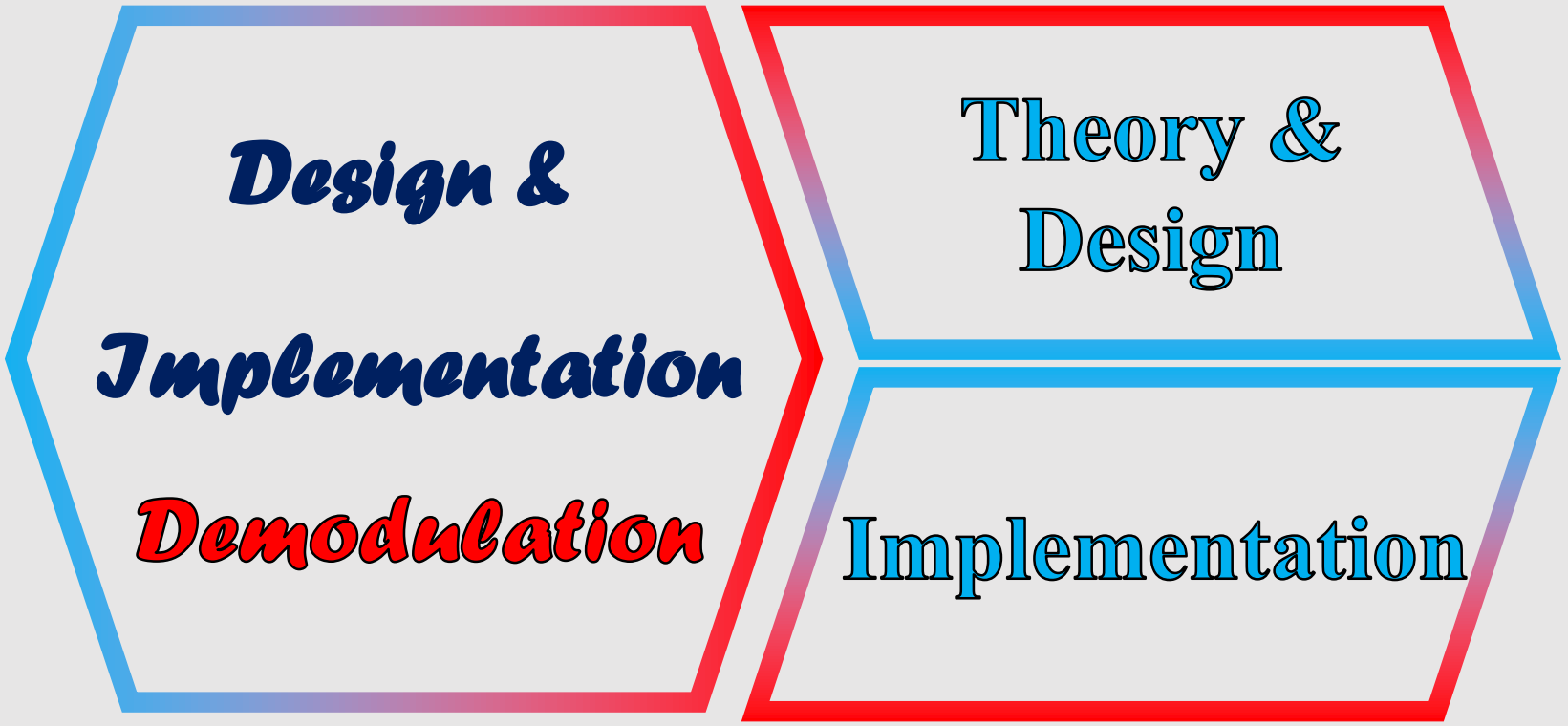
Nov 2022

Content



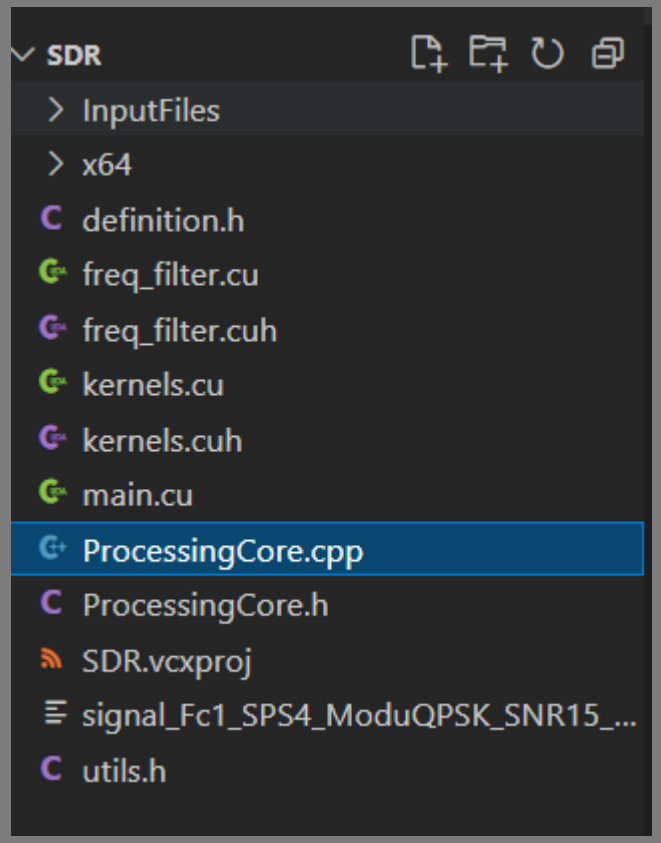


Purpose of The Project

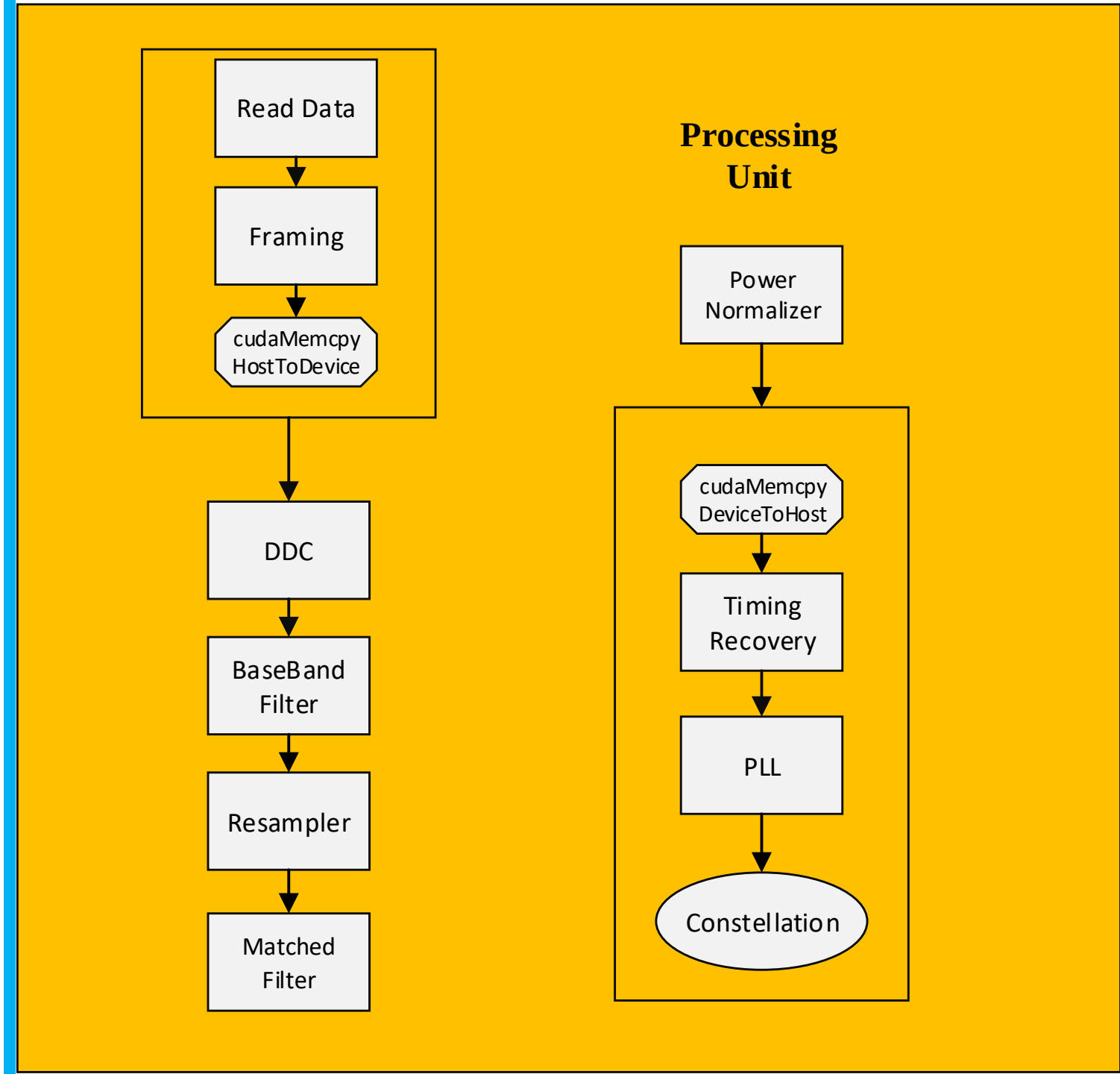

Design &
Implementation
Demodulation
Theory &
Design
Implementation

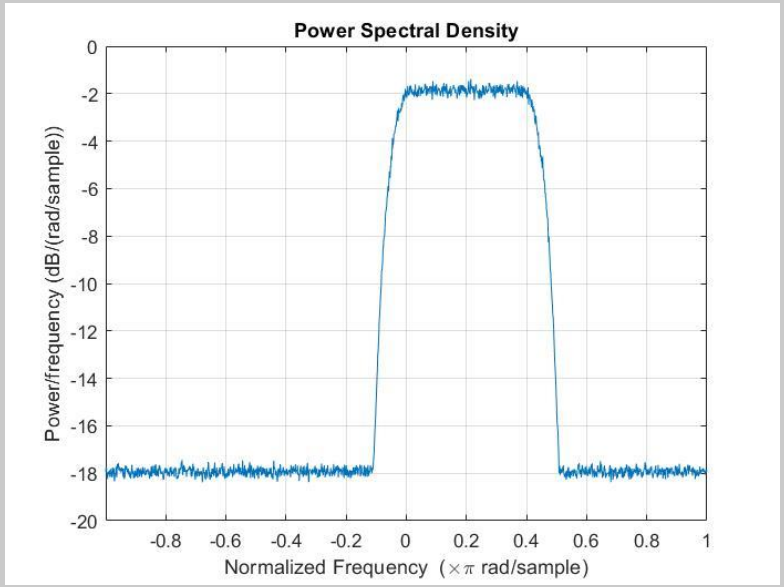
System

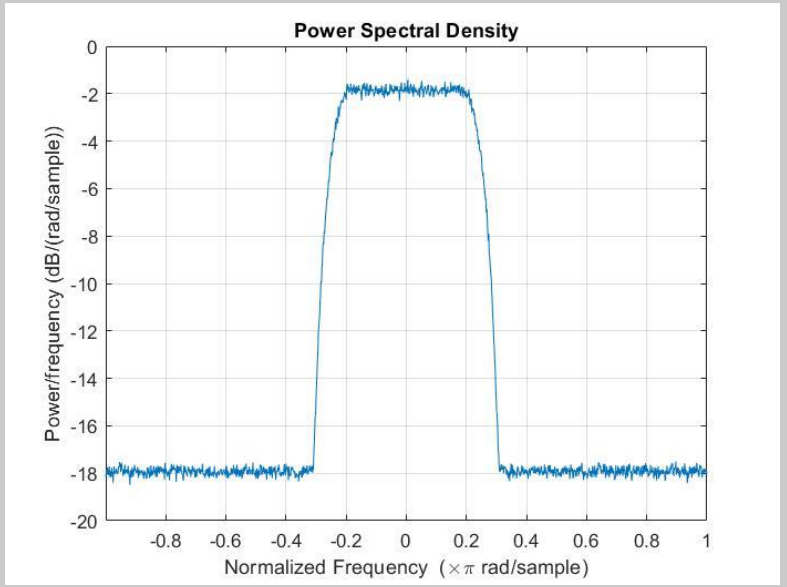
Processing Blocks



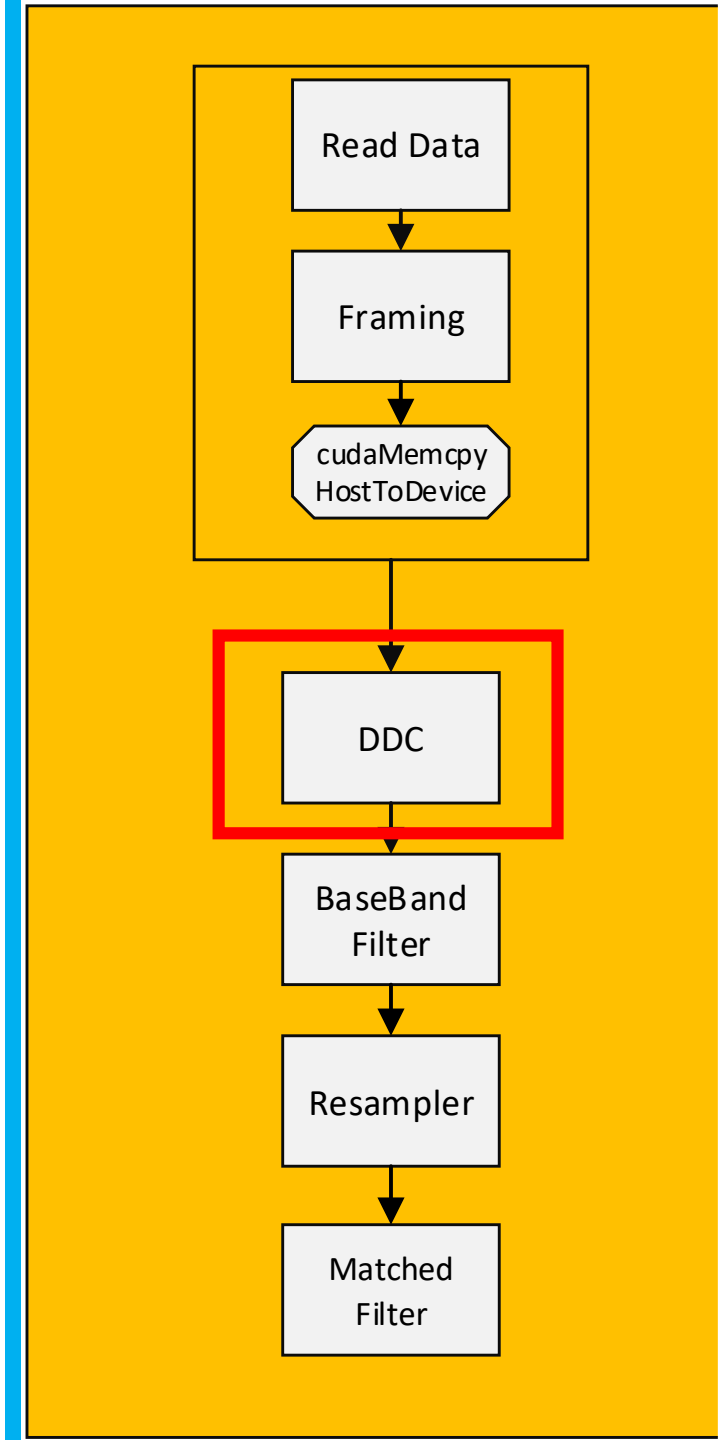
Processing Blocks





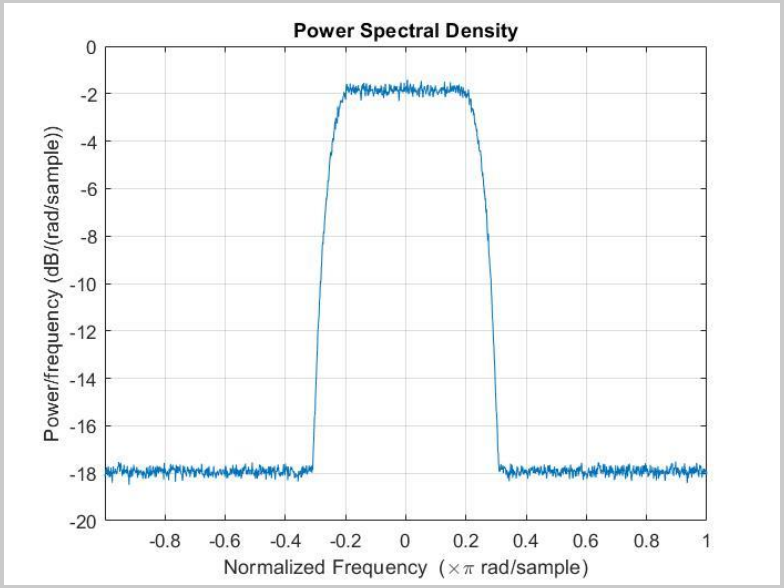


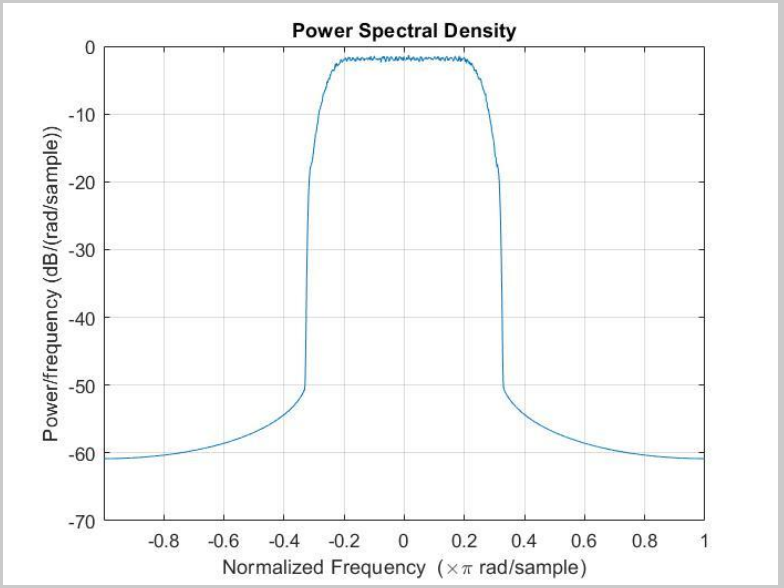
$$y[n] = x[n] * e^{-i2\pi n f_0}$$

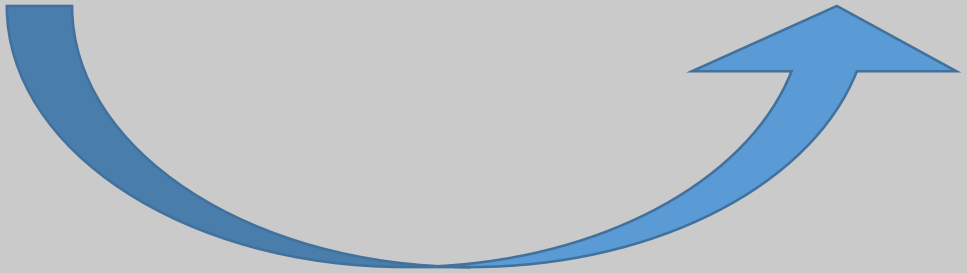


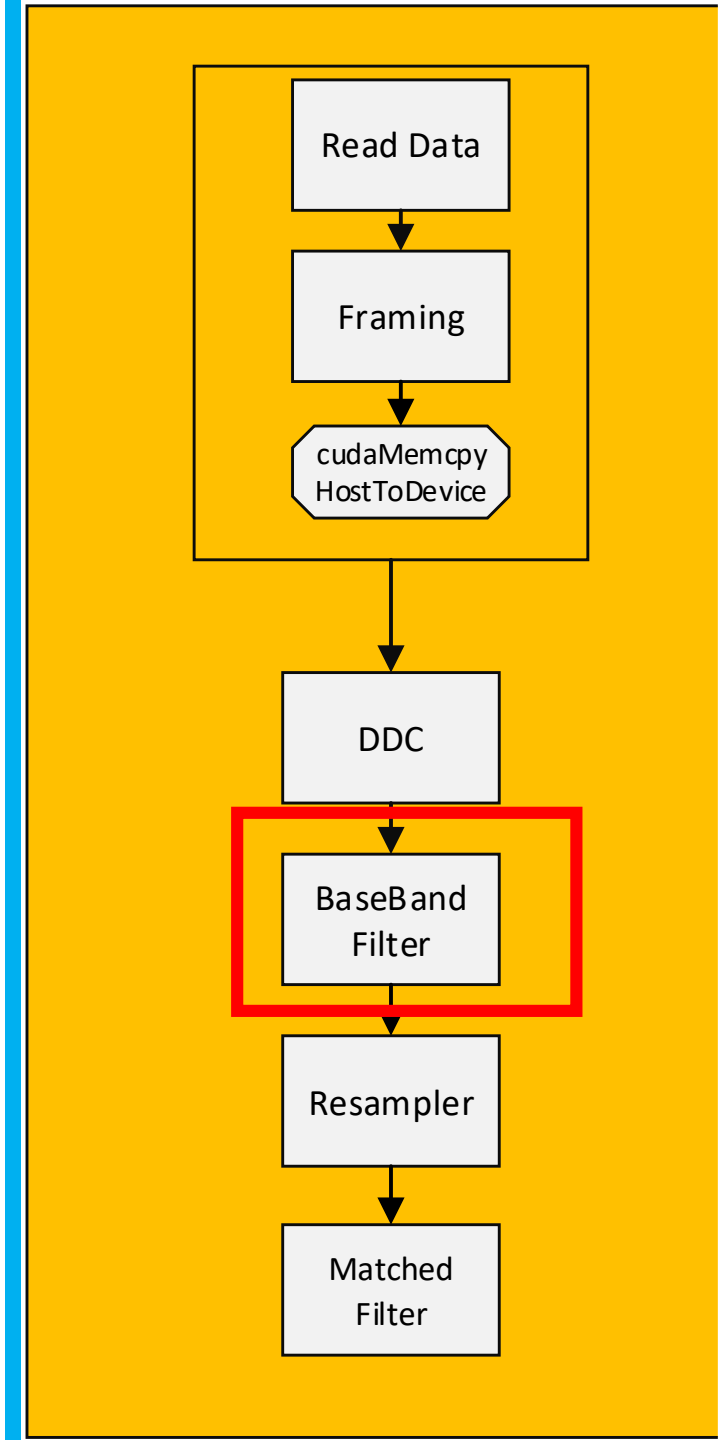
Down Convertor

```
#define PI 3.14159265359f

__global__
void Baseband(float *d_data_Re, float *d_data_Im,
    float *d_DDC_Re, float *d_DDC_Im,
    const int dataLength, const float frequency)
{
    int i = (blockDim.x * blockIdx.x) + threadIdx.x;
    while (i < dataLength)
    {
        float freq = float(i % 10000) * frequency;
        float omega = 2.0 * PI * (freq - int(freq)) + freq_init;
        d_DDC_Re[i] = d_data_Re[i] * cosf(omega) + d_data_Im[i] * sinf(omega);
        d_DDC_Im[i] = d_data_Im[i] * cosf(omega) - d_data_Re[i] * sinf(omega);
        i += blockDim.x * gridDim.x;
    }
}

__global__
void SetFreq_init(const int SOF, float freq)
{
    float freqLast = float(SOF % 10000) * freq;
    freq_init += 2.0*PI*(freqLast - int(freqLast));
}
```


Power Spectral Density
Power/frequency (dB/(rad/sample))
Normalized Frequency (×π rad/sample)


Power Spectral Density
Power/frequency (dB/(rad/sample))
Normalized Frequency (×π rad/sample)




Read Data
Framing
cudaMemcpy
HostToDevice
DDC
BaseBand
Filter
Resampler
Matched
Filter

BaseBand Filter

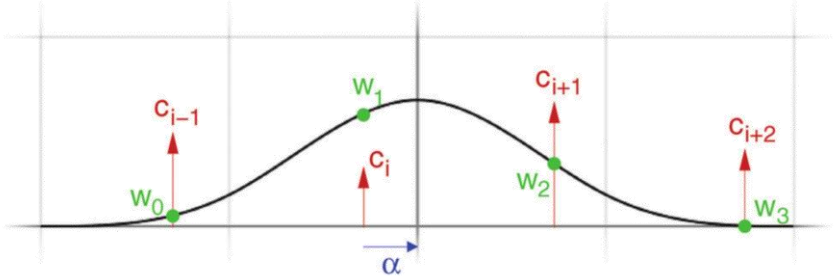
Data Stream Filtering

Cufft

Ovelap Add Method

Using Constant Memory

Resample Filter



for m+3 equally spaced knot points K_j for $j \in \{-3, -2, \ldots m-1\}$ and corresponding knot coefficients τ_j

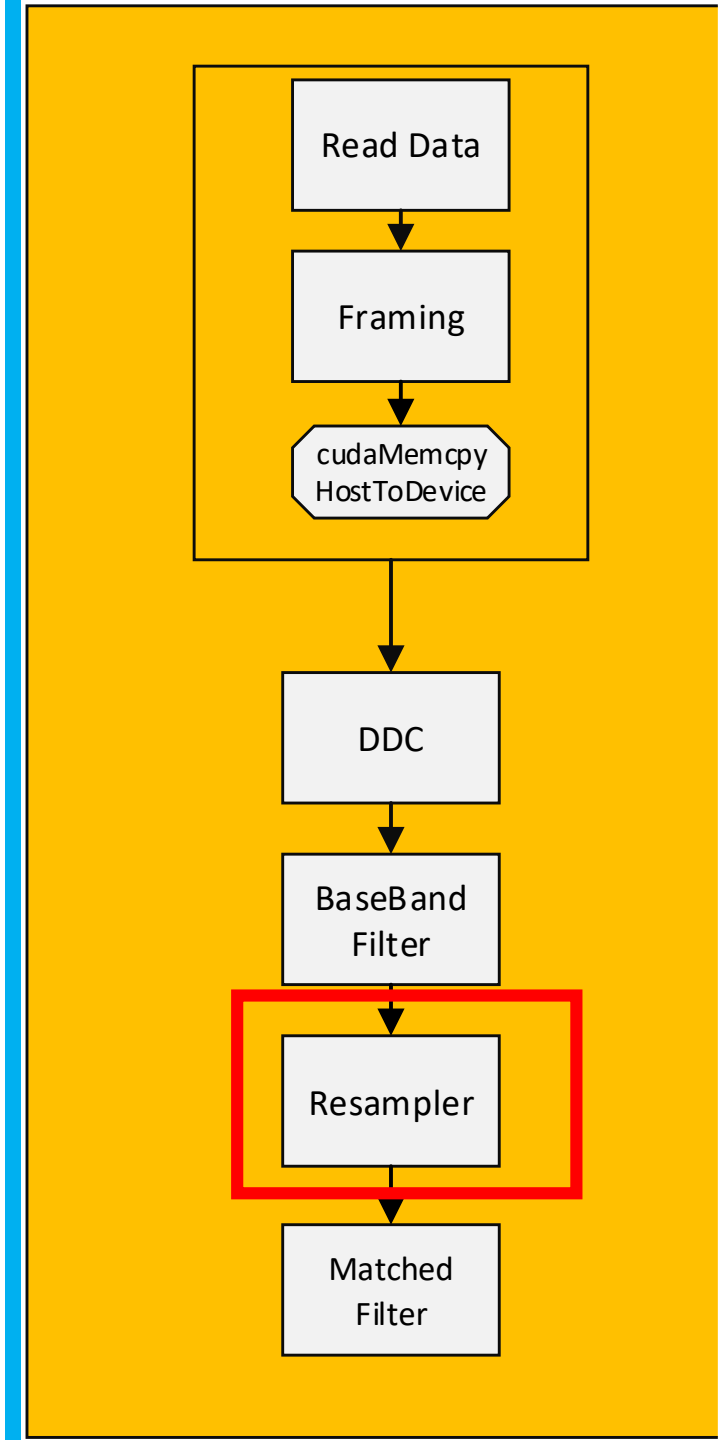
$$S(x) = \sum_{j=-3}^{m-1} \tau_j B_3(\alpha(x - K_j))$$

where, for $t = \alpha(x - K_j)$ and $\alpha = \dfrac{1}{K_j - K_{j-1}}$,

$$B_3(t) = \begin{cases} \text{if } t \in [0,1): \ N_3(t) & \text{for } N_3(t) = \frac{1}{6}t^3 \\ \text{if } t \in [1,2): \ N_2(t-1) & \text{for } N_2(t) = \frac{1}{6}(1 + 3t + 3t^2 - 3t^3) \\ \text{if } t \in [2,3): \ N_1(t-2) & \text{for } N_1(t) = \frac{1}{6}(4 - 6t^2 + 3t^3) \\ \text{if } t \in [3,4): \ N_0(t-3) & \text{for } N_0(t) = \frac{1}{6}(1 - 3t + 3t^2 - t^3) \\ \text{otherwise } 0 \end{cases}$$

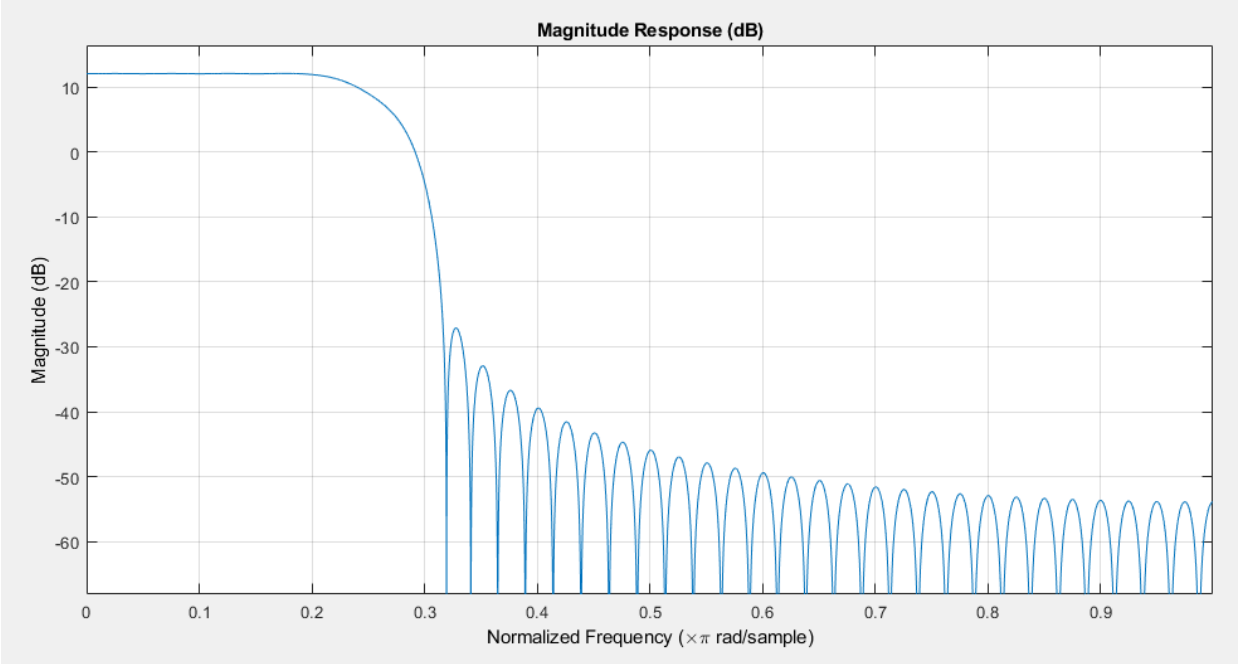
```
__global__ void resampler(float *out_re, float *out_im, int *outLen, float *in_re, float *in_im,
    const float iFs, const float oFs, const int frameLen) {
    float decimationRate{ iFs / oFs };
    int startInterpInputIndex{ 1 - d_resample_nFromPrev };
    float startInterpIndex{ d_last_startInterpIndex };
    float outlen_f{ ((frameLen + 4) - startInterpIndex - 2) / decimationRate };
    int outlen;
    if (outlen_f - floorf(outlen_f) > 0)
        outlen = floorf(outlen_f) + 1;
    else
        outlen = floorf(outlen_f);

    int i{ threadIdx.x + blockIdx.x * blockDim.x };
    float location_float{ (i * decimationRate + startInterpIndex) - floorf(i * decimationRate + startInterpIndex) },
    int interpInputIndex{ static_cast<int>(i * decimationRate + startInterpIndex) }, locStep_int{ d_locStep_int };

    int resample_nFromPrev_local{ d_resample_nFromPrev };
    float interpIndexPos{}, interpClockPhase{};

    //if blockDim.x < 4 take this to the while loop
    if (i < 4) {
        in_re[i] = d_resample_prev_re[i];
        in_im[i] = d_resample_prev_im[i];
    }
    __syncthreads();
```

```
    while (i < outlen) {
        resampler_cubic_interp(in_re + interpInputIndex - 1, in_im + interpInputIndex - 1, location_float, out_re + i, out_im + i);

        if (i == (outlen - 1))
        {
            d_last_startInterpIndex = decimationRate + interpInputIndex + location_float - frameLen;

            for (int tmp{ 0 }; tmp < 4; tmp++) {
                d_resample_prev_re[tmp] = in_re[frameLen + tmp];
                d_resample_prev_im[tmp] = in_im[frameLen + tmp];
            }
            outLen[0] = outlen;
        }

        location_float += locStep_float;
        interpInputIndex += locStep_int + floorf(location_float);
        location_float -= floorf(location_float);

        i += blockDim.x * gridDim.x;
    }
}
```

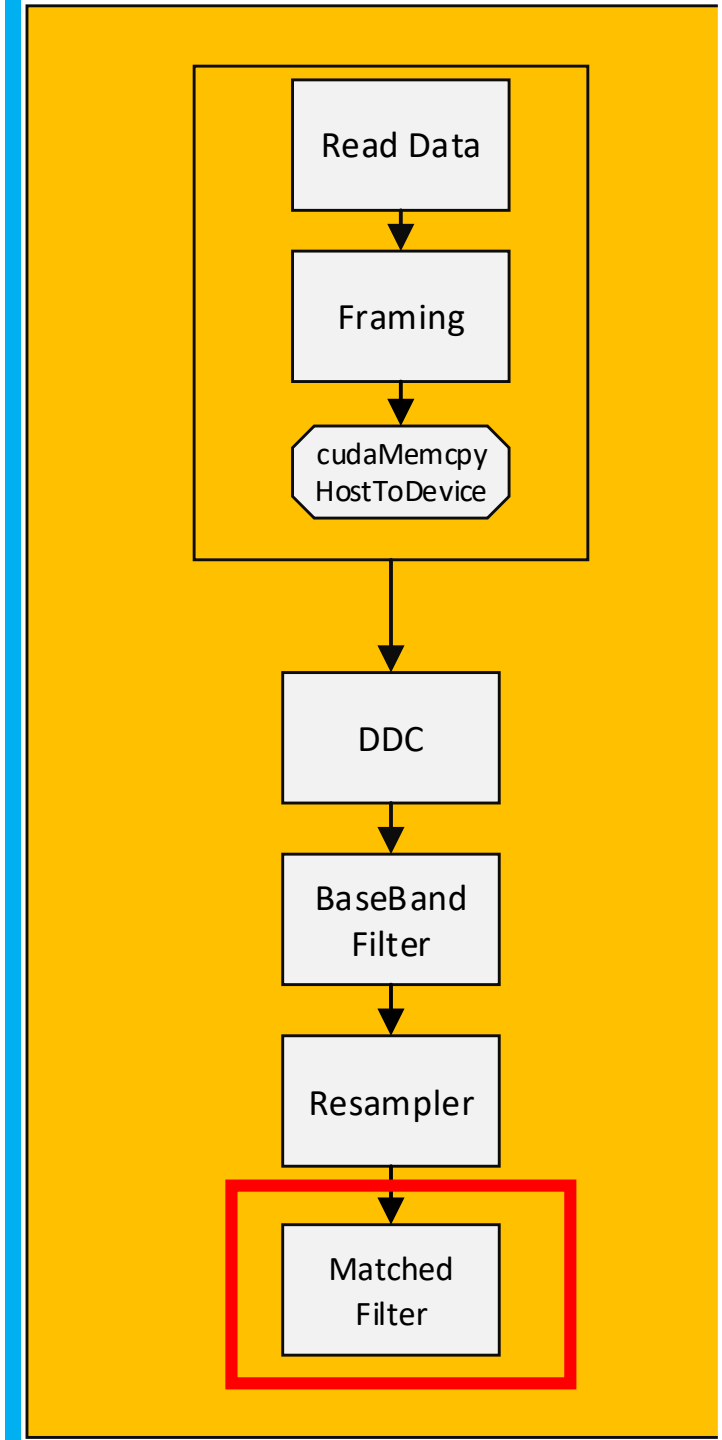


Matched Filter

$$H_{RC}(f)=\begin{cases}1 & \text{for } |f|<2W_0-W \\ \cos^2\left[\frac{\pi}{4}\frac{|f|+W-2W_0}{W-W_0}\right] & \text{for } 2W_0-W<|f|<W \\ 0 & \text{for } |f|>W\end{cases}$$



```cpp
__global__
void MatchFilter(float *d_data_Re, float *d_data_Im, float *d_filteredData_Re,
    float *d_filteredData_Im, float *d_ABS, const int filterLength, const int dataLength)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    float sum_re{ 0 }, sum_im{ 0 };
    while (i < dataLength)
    {
        sum_re = 0.0f;
        sum_im = 0.0f;

        for (int j = 0; j < filterLength; j++)
        {
            sum_re += c_matchFilter_Coef[j] * d_data_Re[i - j + filterLength - 1];
            sum_im += c_matchFilter_Coef[j] * d_data_Im[i - j + filterLength - 1];
        }

        d_filteredData_Re[i] = sum_re;
        d_filteredData_Im[i] = sum_im;

        d_ABS[i] = (sum_re*sum_re + sum_im*sum_im);

        i += (blockDim.x * gridDim.x);
    }
}
```



Power

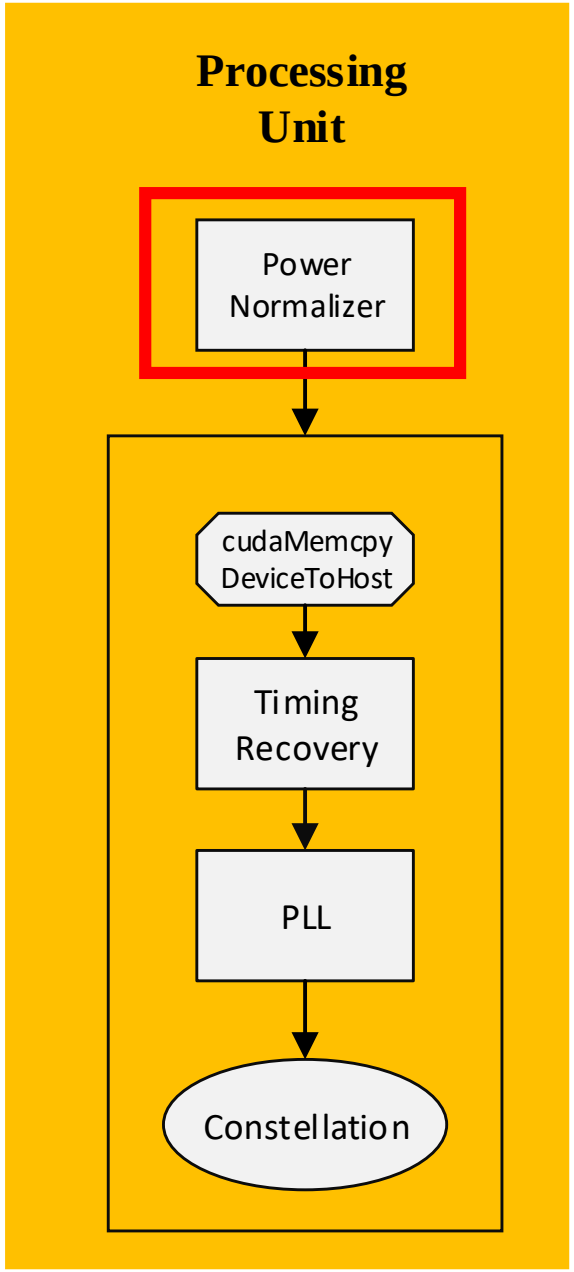
Normalizer

Summation of Abs^2

Shared Memory

Atomic Add

Managed Memory



```cpp
__global__ void parallelSum_arbitraryLen(float *a, float* sum, const int dataLength) {

    int i{ threadIdx.x + blockDim.x * blockIdx.x };
    if (i >= dataLength)
        return;

    int i_thr{ threadIdx.x };
    int prevPowOf2{ 1024 };

    if ((i / 1024 == gridDim.x - 1) && (dataLength % 1024 != 0)) {
        prevPowOf2 = powf(2, floorf(log2f(dataLength % 1024)));
        if (i_thr < dataLength % 1024 - prevPowOf2)
            a[i] += a[i + prevPowOf2];
        __syncthreads();
    }

    int numActive{ prevPowOf2 / 2 };
    for (int j{}; j < log2f(prevPowOf2); j++) {
        if (i_thr  < numActive)
            a[i] += a[i + numActive];
        numActive /= 2;
        __syncthreads();
    }
    if (i_thr == 0) {
        atomicAdd(sum, a[blockDim.x*blockIdx.x]);
    }
}
```

```cpp
__global__
void NF_LoopFilter_MF(float *d_sum, const int dataLen)
{
    int denum{ dataLen };
    d_NormalizeFactor_MF = sqrtf(d_sum[0] / denum);
    if (d_NormalizeFactor_MF == 0)
        d_NormalizeFactor_MF = 1;

    d_sum[0] = 0;
}

__global__
void Normalize_MF(float *d_data_Re, float *d_data_Im, float *d_normalize_Re, float *d_normalize_Im,
    const int dataLen)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int nOut{ dataLen };
    while (i < nOut)
    {
        d_normalize_Re[i] = d_data_Re[i] / d_NormalizeFactor_MF;
        d_normalize_Im[i] = d_data_Im[i] / d_NormalizeFactor_MF;
        i += blockDim.x * gridDim.x;
    }
}
```

Power Normalization

Thank you

The End