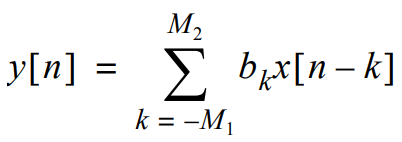
### **Introduction to FIR :**

In signal processing, a finite impulse response (FIR) filter is a filter whose impulse response (or response to any finite length input) is of finite duration, because it settles to zero in finite time. If you put in an impulse, that is, a single “1” sample followed by many “0” samples, zeroes will come out after the “1” sample has made its way through the delay line of the filter.

The term finite impulse response arises because the filter output is computed as a weighted, finite term sum, of past, present, and perhaps future values of the filter input, i.e.,



where both M1 and M2 are finite.

An FIR filter is designed by finding the coefficients and filter order that meet certain specifications, which can be in the time domain (e.g. a matched filter) and/or the frequency domain (most common). Matched filters perform a cross-correlation between the input signal and a known pulse shape. The FIR convolution is a cross-correlation between the input signal and a time-reversed copy of the impulse response. Therefore, the matched filter's impulse response is "designed" by sampling the known pulse-shape and using those samples in reverse order as the coefficients of the filter.

One of the simplest FIR filters we may consider is a 3-term moving average filter of the form

y[n] = (x[n+1] + x[n] + x[n-1])

An FIR filter is based on a feed-forward difference equation as demonstrated by example above

* Feed-forward means that there is no feedback of past or future outputs to form the present output, just input related terms.

Impulse response is the reaction of any dynamic system in response to some external change. E.g. a ball suspended by a spring, when wacked with a bat, responds with a harmonic motion of gradually decreasing amplitude.

### **Proposed method :**

The above 3-term moving average filter requires 3 terms from the input i, i-1, and i+1 for the output without any feedback of previous or future outputs. Parallelization here will be easy due to less dependency. The length of array is taken to be 2048 \* 2048. Further for the first and last discrete value, separate calculation is done use condition. The signal array is divided into blocks of size 512. Then the simple mean filter is applied for one thread where the data can be communicated between the threads easily.

### **Appendix - C Pseudo code :**

/\* This code hasn't been tested, just a pseudo code as per my CUDA understanding \*/

/\*

High Performance Computing Project

Finite Impulse Response using CUDA

- Akshay Kumar (CED15I031)

The FIR function under consideration is

y[n] = (x[n+1] + x[n] + x[n-1]) / 3

\*/

#include <stdio.h>

#include <stdlib.h>

#include <linux/cuda.h>

#define SIZE (2048 \* 2048) // Total size of the Arrays

#define THREADS\_PER\_BLOCK 512 // Number of Threads per block

// kernel code for getting mean filter for the thread

\_\_global\_\_ void mean\_filter (int \*x, float \*y, int n)

{

int index = threadIdx.x + blockIdx.x \* blockDim.x;

if (index < n) // Avoid accessing beyond the end of the arrays

{

if (index == 0) // Exception first element

y[index] = (x[index] + x[index + 1]) / 3;

else

if (index == n - 1) // Exception last element

y[index] = (x[index - 1] + x[index]) / 3;

else

y[index] = (x[index - 1] + x[index] + x[index + 1]) / 3;

}

}

int main (void)

{

// host copies of the arrays x, y

int \*host\_x;

float \*host\_y;

// device copies of the arrays x, y

int \*dev\_x;

float \*dev\_y;

// sizes of both arrays

int size\_x = SIZE \* sizeof (int);

int size\_y = SIZE \* sizeof (float);

// Allocate space for device copies of x, y

cudaMalloc ((void \*\*) &dev\_x, size\_x);

cudaMalloc ((void \*\*) &dev\_y, size\_y);

// Allocate space for host copies of x, y

host\_x = (int \*) malloc (size\_x); random\_ints (x, SIZE);

host\_y = (float \*) malloc (size\_y);

// Copy inputs to device

cudaMemcpy (dev\_x, host\_x, size\_x, cudaMemcpyHostToDevice);

// Launch the mean filter kernel code

mean\_filter <<<(SIZE + THREADS\_PER\_BLOCK -1) / THREADS\_PER\_BLOCK, THREADS\_PER\_BLOCK>>> (dev\_x, dev\_y, SIZE);

// Copy results back to host

cudaMemcpy (host\_y, dev\_y, size\_y, cudaMemcpyDeviceToHost);

// Print input and output

printf ("|\*\*\*\*\*\*\*\*\*\*INPUT\*\*\*\*\*\*\*\*\*\*||\*\*\*\*\*\*\*\*\*OUTPUT\*\*\*\*\*\*\*\*\*\*|\n");

for (int i = 0; i < SIZE; i++)

{

printf ("| x [%6d] = %10.2f || y [%6d] = %10.2f |\n", i, \*(host\_x + i), i, \*(host\_y + i));

}

// Cleanup

free (host\_x);

free (host\_y);

cudaFree (dev\_x);

cudaFree (dev\_y);

return 0;

}