CSE565M: Acceleration of Algorithms in Reconfigurable Logic

Learn by Doing: DFT (Pt. 3)

Anthony Cabrera

FL24::L09

Washington University in St. Louis

Table of contents

Putting it all together (dft.c)

Recall the formulation for the DFT

An N point dft can be determined through a $N \times N$ matrix multiplied by a vector of size N, $G = S \cdot g$ where

$$S = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & s & s^2 & \cdots & s^{N-1} \\ 1 & s^2 & s^4 & \cdots & s^{2(N-1)} \\ 1 & s^3 & s^6 & \cdots & s^{3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & s^{N-1} & s^{2(N-1)} & \cdots & s^{(N-1)(N-1)} \end{bmatrix} \quad (1)$$

and $s = e^{\frac{-j2\pi}{N}}$. Thus the samples in frequency domain are derived as

$$G[k] = \sum_{n=0}^{N-1} g[n] s^{kn} \text{ for } k = 0, \ldots, N-1 \quad (2)$$

Baseline Implementation Considerations

- Must handle complex numbers
- Need to handle data types besides integers, e.g., `float` and fixed point
- How to scale for N-point DFTs for large N
  - for example, it's prohibitive to hold entire coefficient matrix in on-chip memory
- Hence, the body of `data_loop` now has a latency of 6 cycles for each iteration and requires 8 multipliers and 7 adders.
- We went from a latency of $4 \times \text{SIZE} \times \text{SIZE}$ cycles to 6 cycles

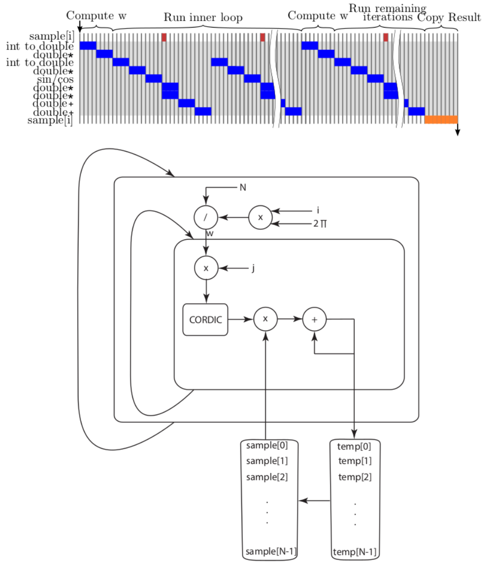
Baseline Implementation

```c
#include <math.h>          //Required for cos and sin functions
typedef double IN_TYPE;    // Data type for the input signal
typedef double TEMP_TYPE;  // Data type for the temporary variables
#define N 256              // DFT Size

void dft(IN_TYPE sample_real[N], IN_TYPE sample_imag[N]) {
  int i, j;
  TEMP_TYPE w, c, s, w_p;
  // Temporary arrays to hold the intermediate frequency domain results
  TEMP_TYPE temp_real[N], temp_imag[N];
  // Calculate each frequency domain sample iteratively
        // (2 * pi * i)/N
  w = (2.0 * 3.141592653589  / N) * (TEMP_TYPE);
  for (i = 0; i < N; i += 1) {
        w_p = i * w;


    // Calculate the jth frequency sample sequentially using HLS sin/cos
    for (j = 0; j < N; j += 1) {

      c = cos(j * w);
      s = -sin(j * w);
      // Multiply the current phasor with the appropriate input sample and keep running sum
      temp_real[i] += (sample_real[j] * c - sample_imag[j] * s);
      temp_imag[i] += (sample_real[j] * s + sample_imag[j] * c);
    }
  }

  // Perform an inplace DFT, i.e., copy result into the input arrays
    // loop interchange optimization
    /*
  for (i = 0; i < N; i += 1)
    temp_real[i] = 0; temp_imag[i] = 0;
```

Implementation Description

- doubly nested `for` loop
  - inner loop multiplies one row of S matrix with input signal sequentially
  - each element of each row of S is converted from phasor to Cartesian coordinates every iteration
  - performs two multiplications for real and imaginary part and accumulates the result
- N iterations for each frequency and N iterations for each point in the FFT leads to $\mathcal{O}(N^2)$ operations
- Reuse the input buffers as the output buffers

DFT Unoptimized Architecture



Can we do better?

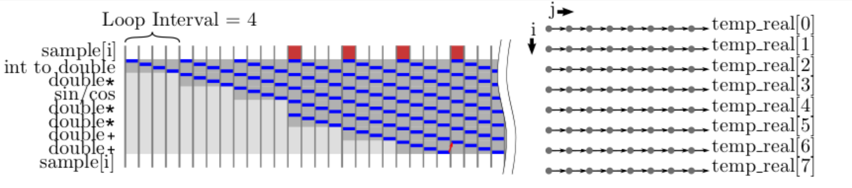
- Reduce the precision of the computation
- Process the data in a different order to pipeline with $II = 1$
- Exploit symmetry of coefficients
- Use different buffers for input and output

Reduce Precision

For example, change from `double` to `float`

Process data in different order

Loop interchange to deal with dependency! This solves the issue of dealing with recurrence dependency.



Exploit symmetry of coefficients

Recall this figure

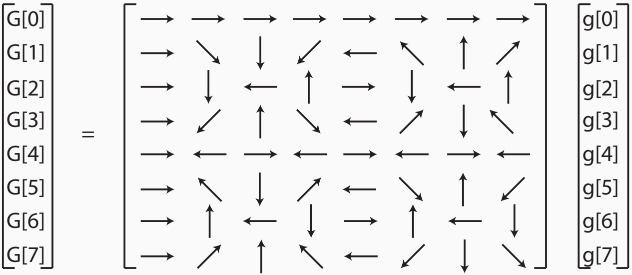


Figure 2: The elements of the *S* shown as a complex vectors.

Use different buffers for input and output

For example, change from `double` to `float`

References i