ECE4703 B Term 2009 Laboratory Assignment 5

Project Code and Report Due by 3:00pm 10-Dec-2009

The goals of this laboratory assignment are:

- to develop an understanding of *frame-based* digital signal processing,
- to familiarize you with computationally efficient techniques that achieve performance gains not through compiler optimization but, rather, through clever *algorithm optimization*, and
- to allow you to experimentally verify the theoretically predicted computational requirements of the FFT and DFT.

1 Problem Statement

The FFT is an efficient algorithm for calculating the DFT and is used in a variety of signal processing applications. The most common implementation method for the FFT is the radix-2 decimation-in-time Cooley-Tukey algorithm where a large DFT is broken down into smaller DFTs by splitting the input samples into odd and even sets. The outputs of the smaller DFTs are reassembled in a special way to form the final result and the overall amount of computation required is much less than a direct calculation of the DFT.

This assignment has three parts. First, you will implement the basic direct DFT, run it in real-time for various values of N, and profile the execution time to observe the computational trends. The second part of the assignment is to implement the radix-2 decimation-in-time Cooley-Tukey FFT, run it in real-time for various values of N, and profile the execution time to observe the computational trends. Finally, you will compare your results to TI's optimized FFT code, primarily in the function `cfftr2_dit.sa`. All math in this assignment should be performed in single-precision floating point and all of your code will be written in C.

2 Part I: Implementation of the DFT

The first part of this assignment is to implement the direct DFT as a function written in C. You will probably want to include the header file `math.h` to allow for pre-computation of the sin and cos terms needed in for the complex "twiddle-factors" of the DFT (do not compute these twiddle factors dynamically in your DFT function as that will be quite slow). Your function must be written generally to allow for any value of N that is an integer power of 2, i.e. $N = 2, 4, 8, 16, 32....$

To facilitate comparisons, you should adhere to the calling convention used by TI's optimized FFT code. Specifically, your function call should work like:

```
* void your_dft( float *x, const float *w, short N)
*
* x Pointer to Array of Dimension 2*N elements holding
* Input to and Outputs from function your_dft()
* w Pointer to an array holding the complex twiddle factors
* N Number of complex points in x
```

Note that everything is globally declared. The input to the DFT is in x and the result of the DFT is returned in x (the input is overwritten by the DFT/FFT function). You should compute the sin/cos portions of the "twiddle factors" prior to the DFT function call and store them in W. All input/output arrays and twiddle factors should be single-precision floating point datatypes.

A suggested flow diagram for the assignment is shown in Figure 1. It is recommended that you use a double buffering technique here since all processing in this assignment is frame-based (your Kehtarnavaz textbook discusses a triple-buffering technique in Lab 6, but the third buffer can be eliminated since the DFT/FFT output overwrites the DFT/FFT input). Let the left channel represent the real part of each input sample and let the right channel represent the imaginary part of each input sample. The (complex-valued) incoming samples are placed in one buffer (with $2N$ `float` elements) and the last complete frame of (complex-valued) samples is stored in another buffer (with $2N$ `float` elements). Your ISR will simply accept new input samples and fill up the incoming buffer; no processing other than incrementing a global index and checking for a full buffer is performed in the ISR. When the incoming buffer is full, you should swap buffers (the "incoming" buffer becomes the "complete" buffer, and vice versa) and begin computing the DFT on the complete buffer in your main code. Your DFT code will compute $2N$ outputs and will overwrite the current "complete" buffer. Your ISR will also be running while you compute the DFT and will be filling up the new incoming buffer. To run in real-time, you need to complete your computation of the DFT before the next incoming buffer is full. You do not need to send any output signals to the AIC23 codec.

Upon completion of the DFT, you should clear any flags that would initiate another DFT and check the status of DIP switch 0. If it has been pressed, you should compute the magnitude of the DFT and store the result in a third buffer. The DFT magnitude buffer will have N `float` elements. You can then plot the magnitude of the DFT output using CCS's plotting capabilities. After you are certain that your DFT is working correctly, profile the execution of your DFT function with and without compiler optimization for various values of N. Increase N until the DFT can no longer execute in real-time and note the maximum value of N that your DFT can handle in real time.

Note that, when DIP switch 0 is *not* pressed, your code should be continuously filling input buffers and computing a DFT each time the buffer is filled. Your code should *not* be computing the magnitude of the DFT unless DIP switch 0 is pressed since computing the magnitude of the DFT will take a lot of extra cycles and will prevent you from increasing N to interesting values. The magnitude function in your code is only to allow for verification that the DFT is working correctly; it should not be running during the normal operation of your code.

To test your DFT, configure the AIC23 codec for 8kHz sampling rate and connect an interesting test signal like a 500Hz square wave to the line-in jack.

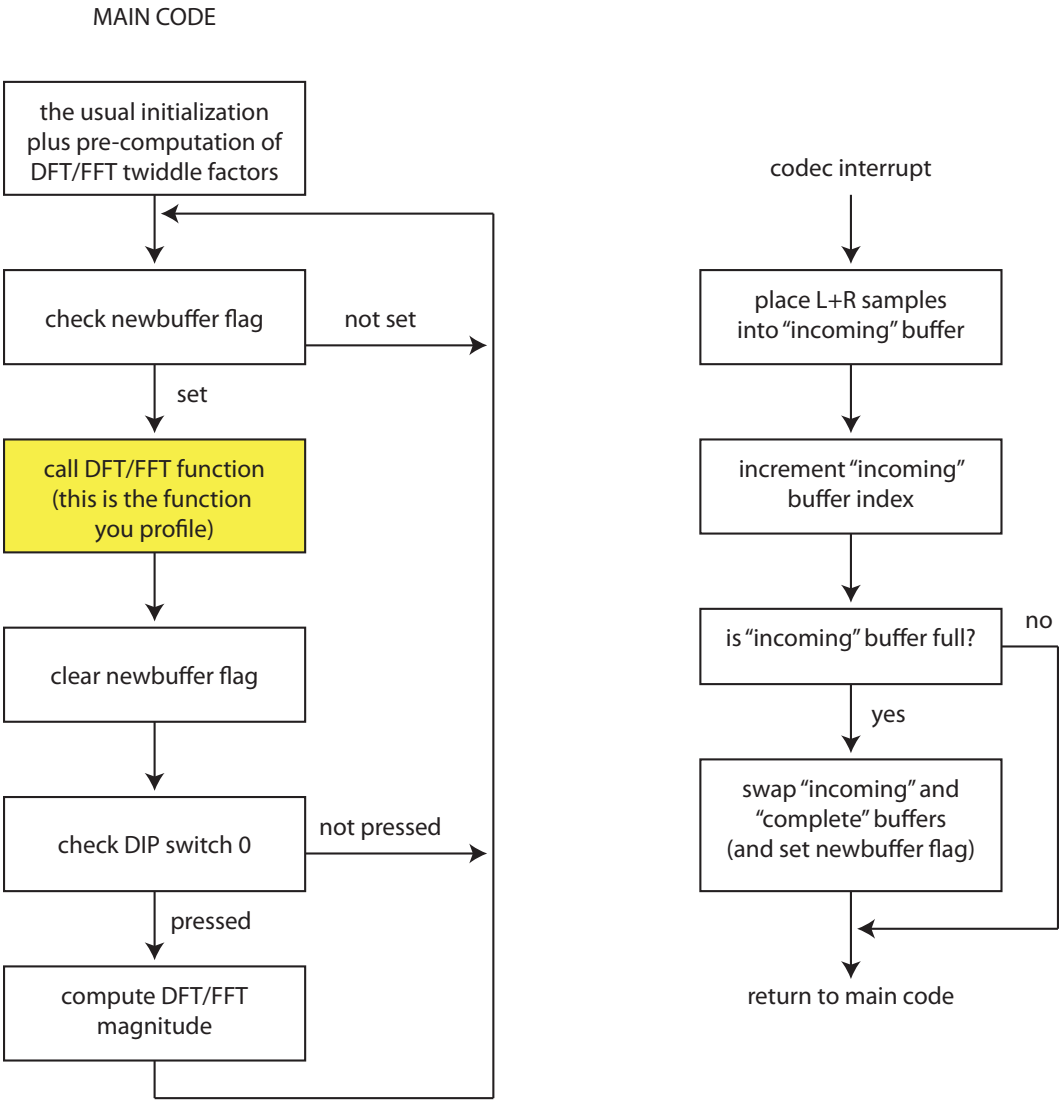

Figure 1: Suggested logical flow of your DFT/FFT code.

3 Part II: Implementation of the FFT

In this part, you will replace your DFT function from Part I with a Cooley-Tukey radix-2 decimation-in-time FFT function. To receive full credit for this part of the assignment, you are required to support at least $N = 2, 4, 8, 16, 32$. It may be tricky to write one general function that works for any value of N, hence it is acceptable to write separate functions, e.g. `fft2`, `fft4`, `fft8`, ... If you choose to take this approach, it is also acceptable to have your higher-numbered FFT functions call the lower number functions, reassembling the outputs of the lower numbered functions appropriately.

Make sure your FFT function is called identically, i.e. has the same function prototype, as the DFT function in Part I. Test and profile your FFT as described in Part I. The output of the FFT should be identical to that of the DFT but, for large enough N, you should see that your FFT executes faster than the DFT.

4 Part III: Using TI's Optimized FFT

Due to the wide variety of applications for the FFT, TI provides an optimized linear assembly function to implement FFTs on the C6x. In this part of the assignment, you will evaluate the performance of TI's routine with respect to your DFT and FFT code.

You will need three functions to use TI's optimized FFT routines. These functions are `cfftr2_dit`, `digitrev_index`, and `bitrev`. These files can be found in various places in the myprojects directory, e.g. the FFTr2 project folder. You should read the header comments in these files to make sure that you know how to use them correctly.

Test and profile TI's FFT as described in Part I. Increase N until the FFT can no longer execute in real-time. Does compiler optimization affect the results?

5 In Lab

You will work with the same lab partner as in the prior laboratory assignments. Please contact the instructor if your lab partner has dropped the course or if you have concerns about your lab partner's performance on the prior assignment.

6 Suggested Procedure for Software Design

1. Begin by at least skimming Chapter 9 and Lab 6 in the Kehtarnavaz text. There are several good examples in here that may give you ideas on how to start the assignment.

2. Make sure your DFT code works before progressing to the FFT. You will need the DFT to check the results of the FFT, so it important that you fully test your DFT code and are confident that it is working correctly. It is recommended that you test your DFT on a buffer with known values and compare the results to Matlab's FFT function. Your DFT function should give exactly the same results as Matlab's FFT function, at least to the precision of the `float` datatype.

3. Write and test your FFT code for smaller values of N first. Recall that, at $N = 2$, the FFT and the DFT perform the same calculations. So you should already have a working `fft2`

function. When writing your FFT code for higher values of N, you can leverage the code you've already written. For example, `fft4` could split the input into odd/even parts and then call `fft2` twice, making sure to correctly reassemble the outputs. Similarly, `fft8` could be written by calling `fft4` twice and correctly reassembling the outputs. You can keep doing this until the FFT no longer runs in real time. Note: It is possible to implement this sort of functionality with recursive function calls (and you are welcome to do so – just be careful about your stack and heap sizes), but recursive function calls are an advanced concept and are not required in this assignment.

4. Make sure your FFT function gives *exactly* the same output as your DFT function. If it doesn't then one (or both) functions are wrong. There are many ways to check your answers: Matlab, Chassaing's or Kehtarnavaz's example code, or even TI's optimized code (which we assume is correct) all can be used to compute a "known good" FFT for comparison.

7 Code Submission and Specific Items to Discuss in Your Report

You can submit one project for the entire assignment where the DFT, FFT, or TI's FFT function is simply selected by changing a `#define` at the top of your main C code. Make sure that there are enough comments to make this obvious to the grader. Your code will be tested for correct functionality and profiled by the grader for select values of N to determine if the profiling results in your report are accurate.

At a minimum, you should include the following results and discussion in your report:

1. A single plot showing

   (a) the exclusive average cycles of your DFT function with and without compiler optimization for $N = 2, 4, 8, 16, 32, ...$

   (b) the exclusive average cycles of your FFT function with and without compiler optimization for $N = 2, 4, 8, 16, 32, ...$

   (c) the exclusive average cycles of TI's optimized FFT function (sum of all three functions required to fully implement the FFT) with and without compiler optimization for $N = 2, 4, 8, 16, 32, ...$

   (d) Predicted trends of $O(N^2)$ and $O(N \log_2(N))$.

   This plot should include distinct line types and a clearly labeled legend. You may want to try generating a semilog or loglog plot to see if that makes your results easier to interpret.

2. Discussion of how the DFT/FFT functions follow (or don't follow) the predicted trends.

3. Analysis and discussion on the largest value of N that can run in real-time in each of the tested cases.

Your report should also discuss any special tricks that you used to implement your FFT code.