ECE4703 B Term 2006 Project 5

Signoff due by 1:50pm 6-Dec-2006. Report due by 4:00pm 7-Dec-2006.

The goals of this laboratory assignment are:

- to familiarize you with computationally efficient processing via the Cooley-Tukey implementation of the Fast Fourier Transform (FFT) and
- to allow you to experimentally verify the computational performance of the FFT versus the 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 Cooley-Tukey algorithm where a large DFT is broken down into smaller DFTs. 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 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 to observe the computational trends. Finally, you will compare your results to TI's optimized FFT code, primarily in the function `cfftr2_dit.sa`.

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 want to include the header file `math.h` to allow for computation of the sin and cos terms needed in the DFT. Your function must be able 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 result of the DFT is returned in x. 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.

To test your DFT, configure the AIC23 codec for 8kHz sampling rate and connect an interesting test signal like a 1kHz sinusoid to the line-in jack. Make the left output channel the real part of $x[n]$ and the right output channel the imaginary part of $x[n]$. Or, alternatively, compute the squared magnitude and output this on either the left or right channels. You can either look at the output using CCS's plotting function or send the output to the AIC23 for display on an oscilloscope. After you are certain that your DFT works, profile the execution of your function with and without compiler optimization for various values of N. Increase N until the DFT can no longer execute in real-time.

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 code is called identically to the DFT code 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, your FFT should execute 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.

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 of 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.

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

4. When writing your FFT code for higher values of N, you may want to leverage the code you've already written. For example, fft8 could split the input into odd/even parts and then call fft4 twice. Similarly, fft4 could then split its input into odd/even parts and then call fft2 twice. It is possible to implement these sort of functionality with recursive function calls, but this is an advanced concept and is not required.

7 Laboratory Report and Grading

See Laboratory Assignment 2.

7.1 Specific Items to Discuss in Your Report

Your report should include, at a minimum, the following results:

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

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

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

Do your profiling results follow the predicted trends of $O(N^2)$ and $O(N \log_2(N))$? How does your FFT code compare to TI's optimized routines, with and without compiler optimization? Your report should also discuss any special tricks that you used to implement the FFT.