ECE6122/6122 **Fall Semester, 2016**

Project 2 – The Threaded Two–Dimensional Discrete Fourier Transform

Assigned: Sept 16, 2016 *Due: October 2, 2016, 11:59pm*

Given a one–dimensional array of complex or real input values of length N, the Discrete Fourier Transform consists af an array of size N computed as follows:

$$H[n] = \sum_{k=0}^{N-1} W^{nk} h[k] \qquad \text{where } W = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \qquad \text{where } j = \sqrt{-1} \quad (1)$$

For all equations in this document, we use the following notational conventions. h is the discrete–time sampled signal array. H is the Fourier transform array of h. N is the length of the sample array, and is always assumed to be an even power of 2. n is an index into the h and H arrays, and is always in the range $0 \ldots (N-1)$. k is also an index into h and H, and is the summation variable when needed. j is the square root of negative one.

An important special case of the above equation is the case of a sample of length 1. Since the summation variable k is exactly 0, the W^{nk} term becomes 1, and thus $H[0] = h[0]$. The Fourier transform of a sample set of length 1 is just the original sample set unmodified.

As in project 1, we are going to compute a two–dimensional Discrete Fourier Transform of a given input image. As before, we first compute the one–dimensional transforms on eacy row, followed by the one–dimensional transforms on each column. However, there are two major differences in this assignment and the prior one.

1. We will use the much more efficient *Danielson–Lanczos* approach for the one–dimensional transforms (described in detail below). This approach has a running time proportional to $Nlog_2N$ as opposed to the N^2 running time of the prior algorithm.

2. We are going to use 16 *threads*, plus the "main" thread, on a single computing platform to cooperate to perform the two–dimensional transform. All threads will have access to the same memory locations (ie. the original 2d image).

3. The main thread will create each of the 16 helper threads and pass the thread id as the argument to the thread function. Each thread will perform a 1D transform on a set of rows (identical to the way we did this with MPI). They should then *barrier* and then do the second 1D transform on the columns. There are a few ways to implement this, so think about a good approach.

Graduate Students. Grad students must implement their own barrier code. Undergrads can use the built-in *PThreads* barrier routines. Also, Grad students should calculate the inverse transform (as we did with the MPI assignment) and save the inverse using `SaveImageDataReal` again as we did with the MPI transform

Description of the Danielson–Lanczos Algorithm

From equation 1, it appears that to compute the DFT for a sample of length N, it must take N^2 operations, since to compute each element of H requires a summation overall all samples in h. However, Danielson and Lanczos demonstrated a method that reduces the number of operations from N^2 to $N\log_2(N)$. The insight of Danielson and Lanczos was that the FFT of an array of length N is in fact the sum of two smaller FFT's of length $N/2$, where the first half–length FFT contains only the even numbered samples, and the second contains only the odd numbered samples.

The proof of the *Danielson–Lanczos Lemma* is quite simple, as follows:

$$\begin{aligned}
H[n] &= \sum_{k=0}^{N-1} e^{-j2\pi kn/N} h[k] \\
&= \sum_{k=0}^{N/2-1} e^{-j2\pi 2kn/N} h[2k] + \sum_{k=0}^{N/2-1} e^{-j2\pi(2k+1)n/N} h[2k+1] \\
&= \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} h[2k] + \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} e^{-j2\pi n/N} h[2k+1] \\
&= \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} h[2k] + W^n \sum_{k=0}^{N/2-1} e^{-j2\pi kn/(N/2)} h[2k+1] \\
H[n] &= H^e_{n \bmod (N/2)} + W^n H^o_{n \bmod (N/2)}
\end{aligned} \tag{2}$$

where H^e_n refers to the n^{th} element of the Fourier transform of length $N/2$ formed from the even numbered elements of the original length N samples, and H^o_n refers to the n^{th} element of the odd numbered samples. Thus the Fourier transform of an array of length N can be computed by summing two transforms of length $N/2$. Then each of the $N/2$ transforms can be computed as the sum of two transforms of length $N/4$, which in turn can be computed as the sum of two transforms of length $N/8$. This process can be repeated until we have a transform of length 1, which we already know can trivially be computed.

Using the equation for the Fourier transform and the Danielson–Lanczos lemma, we can design an easy–to–implement and efficient algorithm for computing the Discrete Fourier Transform for a set of signal samples of length $N = 2^m$. This algorithm is known as the Cooley-Tukey algorithm. By using the Cooley-Tukey algorithm, we can reduce the computational complexity of the DFT computation from N^2 operations to $N\log_2(N)$

Step 1. Reordering the initial h array. Consider a simple case of a sample set of length 8, $h[0], h[1] \ldots h[7]$. Dividing this into the even samples and odd samples, we get:

$$\begin{aligned}
H^e &= h[0] + h[2] + h[4] + h[6] \\
H^o &= h[1] + h[3] + h[5] + h[7]
\end{aligned} \tag{3}$$

Further dividing the even set into it's even and odd components, and similarly dividing the odd set into it's even and odd components, we get:

$$\begin{aligned}
H^{ee} &= h[0] + h[4] \\
H^{eo} &= h[2] + h[6] \\
H^{oe} &= h[1] + h[5] \\
H^{oo} &= h[3] + h[7]
\end{aligned} \tag{4}$$

Finally, dividing each of these into it's even and odd components we get:

$$\begin{aligned}
H^{eee} &= h[0] \\
H^{eeo} &= h[4] \\
H^{eoe} &= h[2] \\
H^{eoo} &= h[6] \\
H^{oee} &= h[1] \\
H^{oeo} &= h[5] \\
H^{ooe} &= h[3] \\
H^{ooo} &= h[7]
\end{aligned} \tag{5}$$

Since each line of equation 5 is just a Fourier transform of length one, we can trivially compute each of these. The question now becomes is there an easy way to determine which of the $h[n]$ values corresponds to each of the H^{xxx}

components. In other words, in equation 5 we determined that H^{eoo} (for example) is equal to $h[6]$, but is there a general way to find this mapping? It turns out that there is, by simply assigning a 0 value to each e (even iteration) and and 1 values to each o (odd iteration), and then interpreting the resulting binary value *in reverse order*. In our example of H^{eoo}, the binary value is 011, which is the value 6 when read from right to left. The first step of the Cooley–Tukey algorithm is to simply transform the original h array from natural ordering ($h[0], h[1], h[2], \ldots h[N-1]$) to the bit reversed ordering. For an original vector of length 8, the reverse ordering is the sequence shown in equation 5. For original array lengths other than 8, the reversed orderings are of course different, but always easy to compute.

After reordering the original h array, we can easily compute the four sets of Fourier transforms of length 2, since the required elements are adjacent in the reordered array. Referring to equation 4, we see that the *ee* values are $h[0]$ and $h[4]$, which are the first two elements in the reordered array; the *eo* values are $h[2]$ and $h[6]$ which are the next two, and so on. Similarly we can compute the two sets of Fourier transforms of length 4, since the *e* elements are the first four and the *o* elements are the next four. Thus the bookkeeping needed for determining which elements of the h array are needed at each step is quite simple.

Step 2. Precomputing the W^n values. Recall the definition of the complex *Weight* factor from equation 1 is:

$$\begin{aligned} W &= e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N) \\ W^n &= e^{-j2\pi n/N} = \cos(2\pi n/N) - j\sin(2\pi n/N) \end{aligned} \tag{6}$$

Further recall that we need the value W^n in equation 2 to combine the *even* and *odd* sub–transforms. Since n is the array index in the original h array (of length N), there are exactly N distinct W^n values ($W^0, W^1, \ldots W^{N-1}$). Since these are somewhat expensive to compute (two trigonometric functions), we can save some time by precomputing the N distinct weights. As it turns out, we in fact only need to compute the first half of these weights. For any W^n, we can show that:

$$W^{n+N/2} = -W^n \tag{7}$$

The proof of this identity is trivial, and is left as an exercise for the reader. Using this identity, we just need to compute W^n for $n = [0, 1, \ldots (N/2-1)]$.

Step 3. Do the transformation. Once the input array is re–ordered in the bit reversed order and the W values are computed, we can easily perform the transform by starting with a two-sample transform of side–by–side elements in the shuffled array, then computing a four–sample transform with four adjacent elements, continuing until we have an N element transform. The problem is complicated by the fact that we want an *in–place* transformation. We *do not* have a separate H array to store the transformed data; rather the original h array is over–written with the computed H elements.

In our example, we would first do a two-sample transform for each of the four sets of two points: {H[0], H[1]}, {H[2], H[3]}, {H[4], H[5]}, {H[6], H[7]},

Note that above we refer to the elements with the symbol H rather than h, since each of the elements are the result of a one–point transform which we already know is simply the point unmodified. For the first two point transform, from equation 2 we can see that the first set would be:

$$\begin{aligned} H[0] &= H^e_{0 \bmod (2/2)} + W_2^0 H^o_{0 \bmod (2/2)} &= H[0] + W_2^0 H[1] \\ H[1] &= H^e_{1 \bmod (2/2)} + W_2^1 H^o_{1 \bmod (2/2)} &= H[0] + W_2^1 H[1] \end{aligned} \tag{8}$$

although we have to very careful about the W values above. To clarify these weights, we use a new notational convention W_N^k. The subscript N indicates the number of points in the transform and the superscript is of course the power. Recall that we pre–computed the weights in step 2 above, but these precomputed weights were for a eight point transform (in our example). In this step we are doing a two point transform, which apparently will result in different weight factors (since the definition of W in equation 1 has N in the definition). Given this, it appears that we have to re–compute the weights for each transform size $N = 2, 4, ...$ Luckily, this is not the case. If we start with weights computed for an N point transform we can prove that for a x point transform (for all x where $x^m = N$ for some $m > 0$),

$$W_x^k = W_N^{kN/x} \tag{9}$$

The proof of this is left as an exercise. Returning to our example, since we have pre–computed weights for $N = 8$, we use:

$$\begin{aligned} W_2^0 &= W_8^{0(8/2)} = W_8^0 \\ W_2^1 &= W_8^{1(8/2)} = W_8^4 = -W_8^0 \end{aligned} \tag{10}$$

Therefore, assuming we stored our pre–computed weights in array W, for the 2–point transform we end up with:

$$\begin{aligned} H[0] &= H^e_{0 \bmod (2/2)} + W_2^0 H^o_{0 \bmod (2/2)} &&= H[0] + W_2^0 H[1] = H[0] + W[0]H[1] \\ H[1] &= H^e_{1 \bmod (2/2)} + W_2^1 H^o_{1 \bmod (2/2)} &&= H[0] + W_2^1 H[1] = H[0] - W[0]H[1] \end{aligned} \tag{11}$$

We would have similar equations for the other three sets of two–point transforms. Continuing to the four–point transforms, we end up with (for the first set of four for example):

$$\begin{aligned} H[0] &= H^e_{0 \bmod (4/2)} + W_4^0 H^o_{0 \bmod (4/2)} &&= H[0] + W_8^0 H[2] = H[0] + W[0]H[2] \\ H[1] &= H^e_{1 \bmod (4/2)} + W_4^1 H^o_{1 \bmod (4/2)} &&= H[1] + W_8^2 H[3] = H[1] + W[2]H[3] \\ H[2] &= H^e_{2 \bmod (4/2)} + W_4^2 H^o_{2 \bmod (4/2)} &&= H[0] + W_8^4 H[2] = H[0] - W[0]H[2] \\ H[3] &= H^e_{3 \bmod (4/2)} + W_4^3 H^o_{3 \bmod (4/2)} &&= H[1] + W_8^6 H[3] = H[1] - W[2]H[3] \end{aligned} \tag{12}$$

In coding this, we must be careful in that we are using a *transform–in–place*, meaning the output array H is in fact the input array h. Notice that in the above equations, we overwrite $H[0]$ in the first equation, but use it on the right–hand–side in later equations. This means we likely would need one or more temporary variables store the original values inside of this processing loop.

We continue transforming larger and larger sample sets (increasing by a factor of two each time) until our final transform is all N samples and the problem is solved.

Graduate Students. Grad students must also implement and call an inverse transform using the same algorithm and same number of threads. The result of the inverse transform should match the original image.

Copying the Project Skeletons

1. Log into `deepthought19.cc` using `ssh` and your prism log-in name.
2. Copy the files from the ECE6122 user account using the following command:

   ```
   /usr/bin/rsync -avu /nethome/ECE6122/ThreadsTransform2D .
   ```

   Be sure to notice the period at the end of the above command.
3. Change your working directory to `ThreadsTransform2D`

   ```
   cd ThreadsTransform2D
   ```

4. Copy the provided `threadDFT2d-skeleton.cc` to `threadDFT2d.cc` as follows:

   ```
   cp threadDFT2d-skeleton.cc threadDFT2d.cc
   ```

5. Then edit `threadDFT2d.cc` to implement the transform.
   (a) Create 16 threads to work in parallel to compute the two–dimensional DFT.
   (b) Implement and call the one–dimensional DFT for the necessary rows using the Danielson–Lanczos approach in the equations above
   (c) Use a barrier to insure all threads have completed the rows.
   (d) Use the same algorithm to compute the one–dimensional DFT for the necessary columns.
   (e) The main program should wait (using a condition variable) until all threads are done, then save the results in file `MyAfter2D.txt` using the `SaveImageData` method in class `InputImage`.

6. Compile your code using `make` as follows:

   ```
   make
   ```

7. Test your solution with the provided inputs. Testing is done by running your program on deepthought19 (or 17)

   ```
   ./threadDFT2d
   ```

Resources

1. `threadDFT2d-skeleton.cc` is a starting point for your program.
2. `Complex.cc` and `Complex.h` provide a completed C++ object containing a complex (real and imaginary parts) value.
3. `Makefile` is a file used by the `make` command to build threadDFT2d.
4. `Tower.txt` is the input dataset, a 1024 by 1024 image of the Tech tower in black and white.
5. `after1d.txt` is the expected value of the DFT after the initial one–dimensional transform on each row, but before the column transforms have been done.
6. `after2d.txt` is the expected output dataset, a 1024 by 1024 matrix of the transformed values.
7. `after2dInverse.txt` is the expected result after the inverse transformation.
8. `InputImage.cc` and `InputImage.h` that will ease the reading of the input data. This object has several useful functions to help with managing the input data.
   (a) The `InputImage` constructor, which has a `char*` argument specifying the file name of the input image.
   (b) The `GetWidth()` function that returns the width of the image.
   (c) The `GetHeight()` function that returns the height of the image.
   (d) The `GetImageData()` function returns a one-dimensional array of type `Complex` representing the original time-domain input values.
   (e) The `SaveImageData` function writes a file containing the transformed data.
   (f) The `SaveImageDataReal` function writes a file containing the transformed data, real part only. This shold be used to write the final results of the inverse transform (grad students only) since the original image had real parts only, the inverse transform should match and have no imaginary parts.

Turning in your Project. Information about turning in your project will be provided.