FFT on reconfigurable hardware

Steven A. Guccione
Mario J. Gonzalez
Computer Engineering Research Center
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, Texas 78712

ABSTRACT

The Fast Fourier Transform (FFT) algorithm is specified in a data parallel version of 'C'. This specification is used to produce a custom circuit suitable for use in a system based on reconfigurable logic. Performance estimates indicate that this approach is capable of producing the two dimensional Fourier transform of images at real time video rates.

Keywords: FPGA, reconfigurable logic, Fourier transform, signal processing

1. INTRODUCTION

A popular method for producing spectral information about a signal is the Fourier transform. This transform converts a signal in the time domain to one in the frequency domain. Instead of representing a signal by an amplitude which varies over time, the signal is represented as a series of sinusoidal frequency components, each with a phase and magnitude. This representation is sometimes of value for its own sake, and at other times it provides an efficient means of processing and filtering the signal.

The Fourier transform and its implementations have a somewhat colorful history. The concept of representing signals in the frequency domain goes back at least as far as the Babylonians. A brief historical account, with references can be found in Oppenhheim, Willsky and Young.[4]

While a potentially powerful technique for processing signals, calculation of the Fourier transform can be computationally intensive. Various techniques, including sophisticated custom hardware[8] have been employed to produce the Fourier transform. In this paper, the Fourier transform will be implemented for reconfigurable hardware. This implementation will be analyzed and compared to other implementations.

2. THE DISCRETE FOURIER TRANSFORM

While the Fourier transform was originally described in terms of continuous functions, it is also possible to perform a Fourier transform on digitized data. Here, a set of numeric values represent the amplitude of the signal at evenly spaced intervals. Given a set of N values, it is possible to compute the Fourier series which represents the digitized signal in the frequency domain. The result of this calculation is two sets of N values, representing the amplitude and phase of the components in the frequency domain. This transformation is commonly referred to as the *Discrete Fourier Transform* or *DFT* to distinguish it from its continuous counterpart.[1,5]

Equation 1 is used for computing the components of the discrete Fourier transform. The original signal is represented by the N sampled values in h_0 through h_{N-1}. Each of these sampled values is multiplied by a complex value and summed to produce a single component of the frequency domain representation, H_n.

$$H_n = \sum_{k=0}^{N-1} h_k e^{2\pi ikn/N} \tag{1}$$

From this equation we see that each of the N values of the original signal h contribute to the calculation of each value H in the frequency domain. The direct implementation of this equation is clearly of $O(N^2)$ complexity.

Equation 2 shows Equation 1 with the real and imaginary portions of H_n computed separately. The real portion $Re(H_n)$ gives the magnitude of the Fourier components, while the imaginary portion, $Im(H_n)$ gives the phase.

$$Re(H_n) = \sum_{k=0}^{N-1} (Re(h_k)cos(2\pi kn/N) + Im(h_k)sin(2\pi kn/N)) \quad (2)$$

$$Im(H_n) = \sum_{k=0}^{N-1} (Im(h_k)cos(2\pi kn/N) - Re(h_k)sin(2\pi kn/N))$$

From this representation, a data parallel version of the algorithm may be specified. Figure 1 gives one approach to this calculation. This data parallel implementation of the DFT calculates a single component n of the DFT. This calculation must be repeated N times to produce all N components of the DFT.

The first portion of the algorithm generates a vector containing the integers from zero to $(N-1)$. In a serial version of this algorithm, these values for K would typically be found in the control loop which keeps track of the number of terms computed. As with other data parallel calculations, this control structure information is not directly available to the calculation. The integer values for K are instead produced by the first three lines in the algorithm. Note that these three lines could have easily been condensed into a single line of source code of the form $K =$ *add-scan(1) - 1*. It is shown in its expanded form with comments for clarity.

Once the K values are produced, the constant *w0* is calculated. K and *w0* are then multiplied and the *sin()* and *cos()* taken. This code assumes that library implementations of the *sin()* and *cos()* functions already exist. These implementations may consist either of custom designed circuitry or arithmetic sequences in data parallel code.

```
K = 1; /* K = 1, 1, 1, ..., 1 */
K = add-scan(K); /* K = 1, 2, 3, ..., N */
K = K - 1; /* K = 0, 1, 2, ..., (N-1) */

w0 = (2 * PI * n) / N;
Sin = sin(K * w0);
Cos = cos(K * w0);

Dft_re = ((In_re * Cos) + (In_im * Sin));
Dft_im = ((In_im * Cos) - (In_re * Sin));

Dft_re = add-scan(Dft_re) / N;
Dft_im = add-scan(Dft_im) / N;
```

Figure 1. The data parallel code for the DFT.

Once the transcendental functions are computed, the real and imaginary portions of the DFT are calculated. These are summed using the *add-scan()* operation and scaled by a factor of N. The vector calculation of length N produces a single value in the DFT of the signal. This procedure must be repeated N times to produce the complete transform.

From the data parallel code in Figure 1, a pipelined circuit can be extracted.[3] Figure 2 shows the extracted circuit. Note that two vectors are input producing two output vectors. The output vectors, however, contains the summed values. Only the final sum is likely to be of interest.

This implementation was simulated using the data in Figure 3. Thirty-two samples of the function $\cos(2\pi n/8)$ taken at unit intervals are used as the input. The impulses in the figure represent the sampled values while the sinusoidal envelope gives the sampled function.

Theory predicts that the Fourier transform of a cosine function will produce two positive symmetrical impulses. The results of the DFT calculation in Figure 4 verify this expectation.

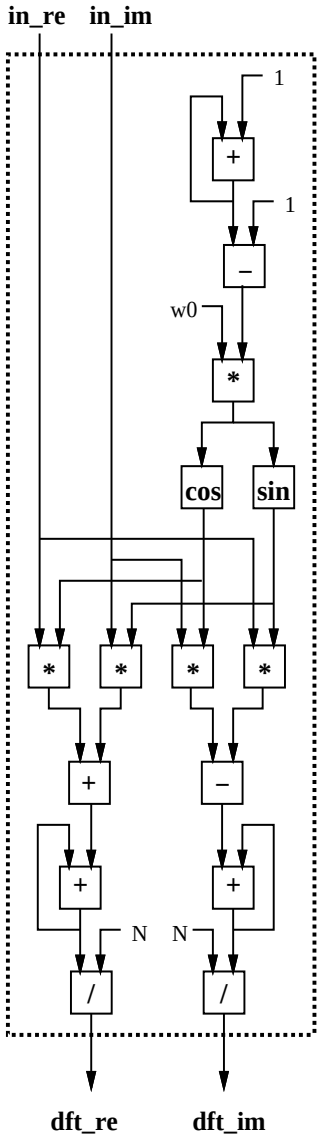

Figure 2. The circuit for the DFT.

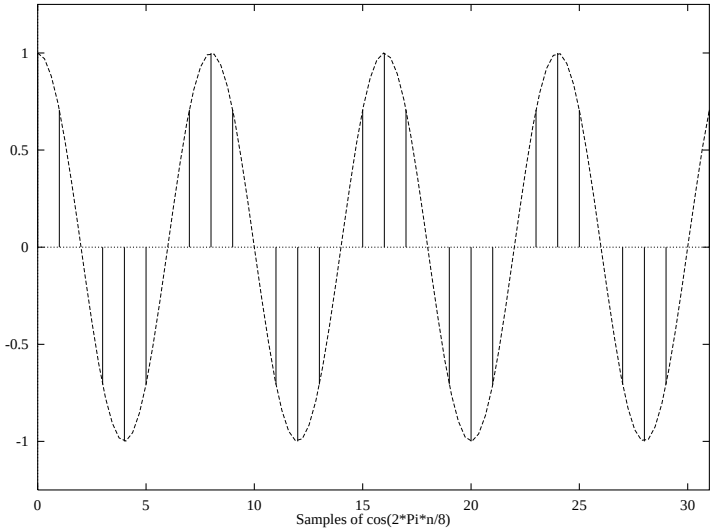

Figure 3. The 32 samples of the function $\cos(2\pi n/8)$.

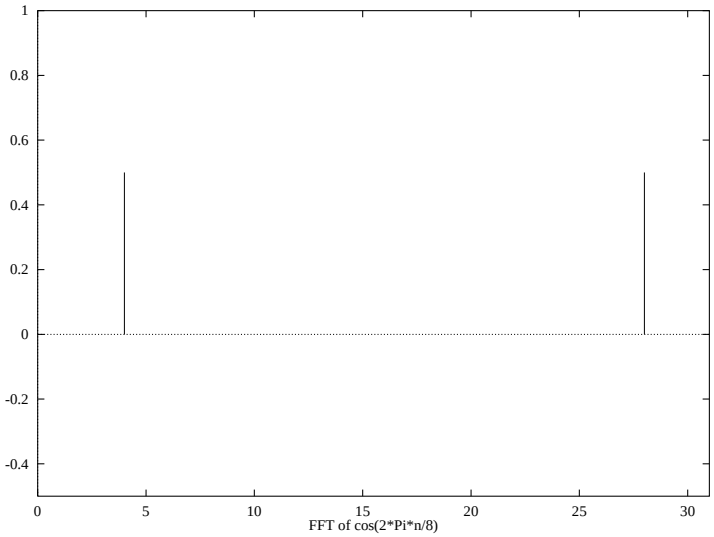

Figure 4. The DFT of the function $\cos(2\pi n/8)$.

2.1. The Fast Fourier Transform

While the representation of a digitized signal provided by the Fourier transform has many powerful uses, generating this frequency domain representation from the sampled time domain data is computationally intensive. The calculation involves all N samples to compute each of the N transform values. This gives the calculation an overall complexity of $O(n^2)$. Additionally, the computation involves repeated computation of sine and cosine values, which are often themselves computationally intensive.

Fortunately, a more efficient method for computing the DFT exists. This method is commonly known as the *Fast Fourier Transform* or *FFT*. The modern version of this algorithm was published by Cooley and Tukey in 1965, although similar methods were reported earlier in the century. The general technique goes back at least as far as Gauss in the early 19th century.

This technique is based on restructuring the computation to take advantage of previously computed values. One way to view the restructured calculation is that of breaking the computation of a single DFT of length N into two DFT calculations of length $N/2$. Equation 3 gives the method used to decompose the DFT into these two calculations.

$$\begin{aligned} H_n &= \sum_{k=0}^{N-1} h_k e^{2\pi i k n/N} \\ &= \sum_{k=0}^{(N/2)-1} h_{2k} e^{2\pi i k n/(N/2)} + W^n \sum_{k=0}^{(N/2)-1} h_{2k+1} e^{2\pi i k n/(N/2)} \\ &= H_n^e + W^n H_n^o \end{aligned} \tag{3}$$

This equation reveals that the two components used to produce the DFT are composed of the even and odd values of the original data, respectively. These two smaller DFT calculations are summed, with the odd portion of the calculation multiplied by the scale factor W^n. This reduces the complexity of the calculation from $O(n^2)$ to $2 * O((n/2)^2)$.

This process can be carried further, with each of the two smaller calculations split into even and odd portions, reducing the complexity to $4 * O((n/4)^2)$. This process may be continued until the data can no longer be divided in two. The final DFT of a single value h_n is simply the value of h_n. For values of N which are an even power of two, this produces an overall complexity of $O(n \log_2(n))$. While this is not a significant improvement for small values of n, for larger values, the improvement can be substantial. For $N = 16$, the FFT is roughly four times faster than the standard DFT. For $N = 1024$, the difference is a factor of 100. Larger N produce even larger gains in efficiency.

In Equation 3, the value W^n is specified as a scale factor for the odd portion of the FFT. These values, sometimes referred to as *twiddle factors* take the complex value given in Equation 4. Note that the computation of W^n contains all of the transcendentals used in the calculation.

$$W^n = e^{2\pi i/N} \tag{4}$$

This method of computing the FFT is often given the graphical representation shown in Figure 5. Here, a small FFT of eight values is computed. Note that even and odd values are paired. Also note that there are three distinct stages of the calculation, providing the logarithmic component of the complexity.

From this representation, a basic *cell* can be viewed as the basic component of the FFT calculation. This cell is shown graphically in Figure 6. This cell takes two complex values, a and b as its input. These input values are used to produce the two complex output values c and d. The value of c is simply the sum of a and b. The value of d is the difference of a and b, multiplied by the appropriate value of W^n.

Curiously, while most descriptions of the FFT rely on a basic FFT cell such as that in Figure 6, the typical figure such as the one in Figure 5 does not clearly distinguish these cells, except in the last stage of the calculation. All other such cells are stretched and interleaved in some fashion. As an alternative to this representation, Figure 7 rearranges the elements of the diagram to provide distinct cells. This rearrangement causes the previously interleaved data paths to be independent units, but causes the values passed between stages to be interleaved.

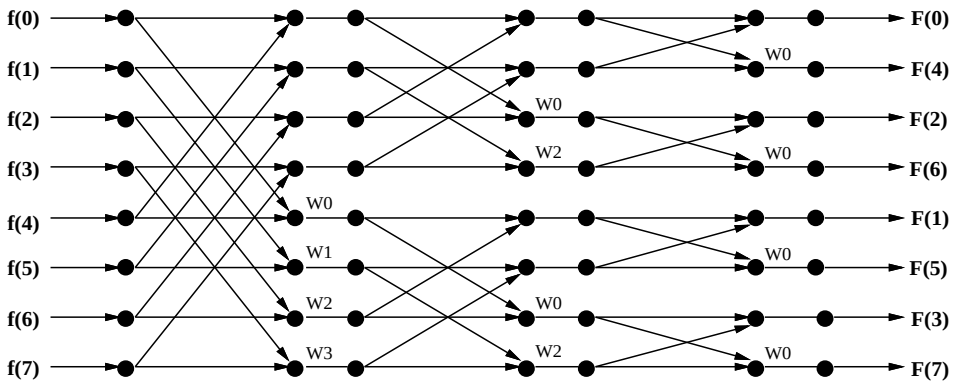

Figure 5. A standard representation of the FFT.

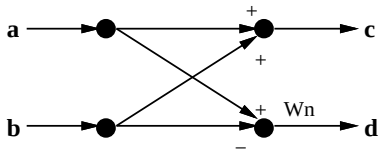

Figure 6. The basic FFT cell.

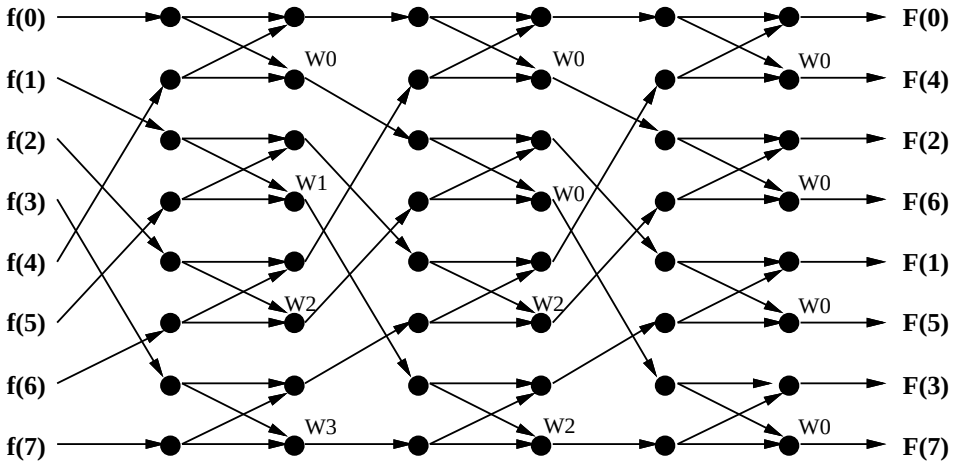

Figure 7. An alternate representation of the FFT.

Fortunately, the interleaving between stages is the same for each stage in the calculation. The form of this interleaving is that of a *perfect shuffle*. The perfect shuffle is so named because it produces data interleaved in a manner similar to that of a deck of playing cards being shuffled. The perfect shuffle is known to have some interesting computational properties and has been studied in other contexts.[7]

From this representation, the implementation of the basic cell is fairly simple, but the data patterns necessary to produce the desired results may require further exploration. Figure 8 shows the way in which vectors of data are accessed to feed the inputs and produce the outputs from the basic FFT cell. Assuming the input values are available in a single vector, producing the vector inputs A and B is fairly simple. The input vector is split in half, with the first $N/2$ elements comprising the A vector and the second $N/2$ elements comprising the B vector.

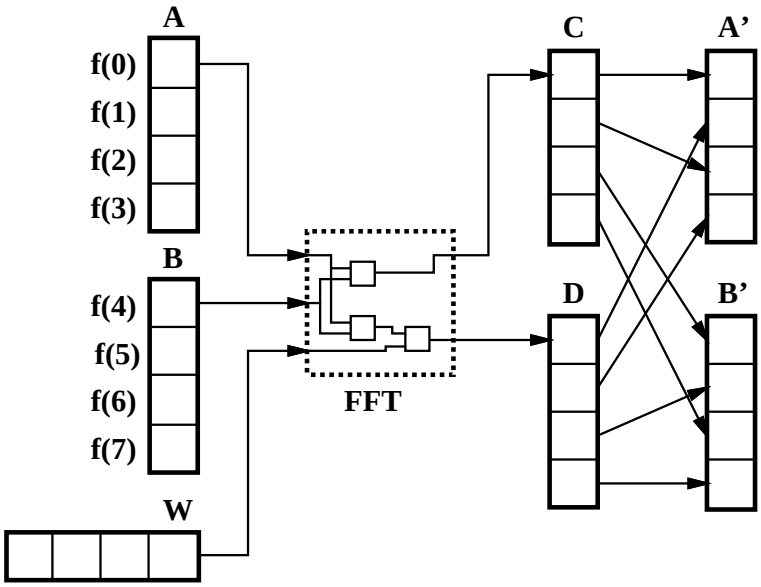

Figure 8. Vectorizing the FFT.

These inputs, along with the appropriate values for W^n produce two output vectors C and D of length $N/2$. Before these vectors can be used by the subsequent stages of the calculation they must be reordered. If C and D are stored in contiguous memory locations, they may be viewed again as a single vector of length N. A perfect shuffle of this vector produces a new vector of length N which may be split into two new vectors A' and B' and fed into the next stage of the calculation.

Fortunately, the perfect shuffle can be easily implemented using vector striding. Two accesses of stride two produce the desired result. It should be emphasized that for a system which has direct hardware support for vector striding, this operation takes no resources. The *stride()* function simply describes the access pattern used. Data are not necessarily physically manipulated by this operation.

With the basic cell and the data access pattern specified, the algorithm can now be implemented using data parallel code. The implementation of the basic cell for complex numbers can be taken directly from the diagram for the cell in Figure 6.

Figure 9 shows the data parallel code for the basic cell. Note that since this calculation involves complex values, both real and imaginary values are computed. For the calculation of the vector C, this is simply two additions. For the subtraction then multiplication by W^n in the calculation of D, the multiplication of two complex values produces a more complicated expression.

The code in Figure 9 is executed on the vectors of length N for the required $\log_2(N)$ iterations to produce the final DFT values. From this data parallel code, the circuit in Figure 10 is extracted.

While this is the bulk of the algorithm, the generation of the values for the W vector have not yet been addressed. The vector W is also a complex quantity with real and imaginary components. Equation 5 shows the expanded version with transcendental functions sine and cosine.

```
C_re = A_re + B_re;
C_im = A_im + B_im;

D_re = (W_re * (A_re - B_re)) - (W_im * (A_im - B_im));
D_im = (W_re * (A_im - B_im)) + (W_im * (A_re - B_re));
```

Figure 9. The data parallel code for the FFT.

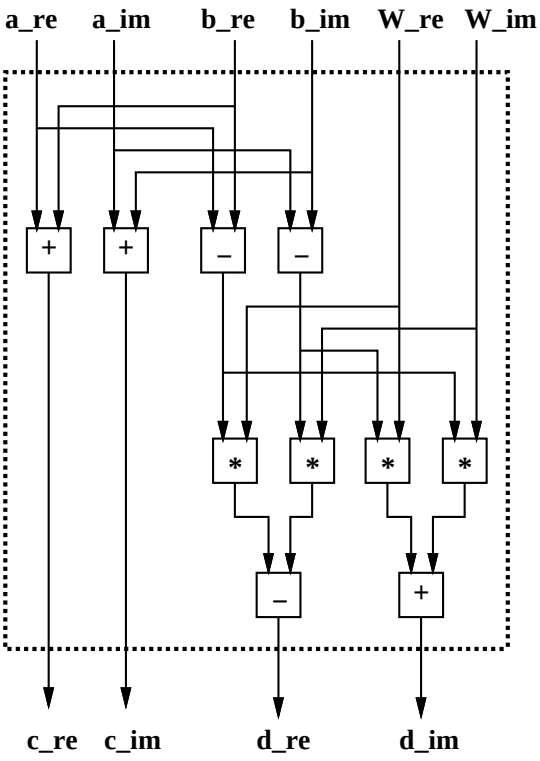

Figure 10. The extracted FFT circuit.

$$W^n = \cos(2\pi n/N) - i\sin(2\pi n/N) \tag{5}$$

The vector W contains $N/2$ values which are reused throughout the calculation. This vector should be precomputed at the beginning of the calculation. While the reconfigurable hardware may easily be employed in this computation, this may or may not be the best approach. Because of the small size of this vector it is possible that the host may be profitably employed in this portion of the computation. Also note that the vector W changes with each stage of the computation. Unfortunately, no simple vector striding access will produce this new W vector. Again, since this operation is fairly simple, it may be desirable to employ the host in this manipulation.

Finally, as with the standard FFT calculation, the data is produced in a peculiar order. This order is often referred to as *bit reversed* order. Here, the actual index of the result is determined by reversing the bits of the vector index of the values produced. For example, in Figure 7 the second transformed value in location "1" is denoted *F(4)*. The index "4", or binary (100) is determined by reversing the bits in the binary value of its actual location, 1, or binary (001).

Unfortunately, these is no obvious data access pattern that would permit vector striding to be used to reorder the data. Again, it is likely that this function would be best performed by the host processor. Note that the solution involves the swapping of pairs of elements in the vector and is a fairly simple process.

The complete algorithm for the FFT using this approach has been implemented and simulated. Using the data from the DFT example in Figure 3, a result identical to Figure 4 is produced. Because the values are identical, they are not reproduced here.

2.2. The FFT in 2D

While the Discrete Fourier Transform is defined to act on a vector of digitized waveform data, it is not strictly limited to one dimensional signal data. The DFT may also be applied to image processing tasks. The use of a Fourier representation for digitized images presents many interesting possibilities for image enhancement. The image processing functions which use a spectral representation are often difficult to perform using a standard image representation.

To produce the two dimensional DFT of a digitized image, the DFT of each row of pixels in the image must be taken. The result of these DFT operations is then processed by taking the DFT of these values by columns, rather than by rows. (For a more detailed analysis, as well as some interesting processing techniques based on digital images represented in the frequency domain, the reader is referred to Gonzalez and Wintz.[2])

For an NxN image, $2N$ transforms are required. If the FFT is employed the complexity of the complete two dimensional image FFT calculation is $O(2N^2 \log_2(N))$. For even moderately sized images, this represents a significant amount of calculation.

The data parallel implementation of the FFT is used to compute the two dimensional FFT of a simple test image. Figure 11 shows a $256x256$ pixel image where each pixel was represented by eight bits, giving 256 distinct shades of grey. The processed image shows the spectral pattern produced by this synthetic image.

It should be noted that this spectral image is displayed in a somewhat unconventional manner. First, the image was translated so that the "interesting" portion of the image is moved to the center. This translation can be viewed as dividing the image into four quadrants and swapping diagonal quadrants. This places the pixels which were originally in the corners at the center of the image.

Second, the values of the pixels are scaled in a logarithmic manner. This is because the values produces by the FFT tend to be logarithmically distributed. If it were not for this scaling, the image would appear more as a small white dot in the center of the frame. Finally, the image represents only the real portion of the processed image. The phase information in the imaginary portion of the processed image, while important, does not contain information which is as meaningful when viewed as an image.

Figure 12 contains a more realistic image and its Fourier transform. Again the transform image is a scaled version of the real portion of the transformed image.

This two dimensional image DFT has some computational features which should be mentioned. First, since all of the transforms are of a fixed length N, the W vector need only be calculated once. The versions of the W vector

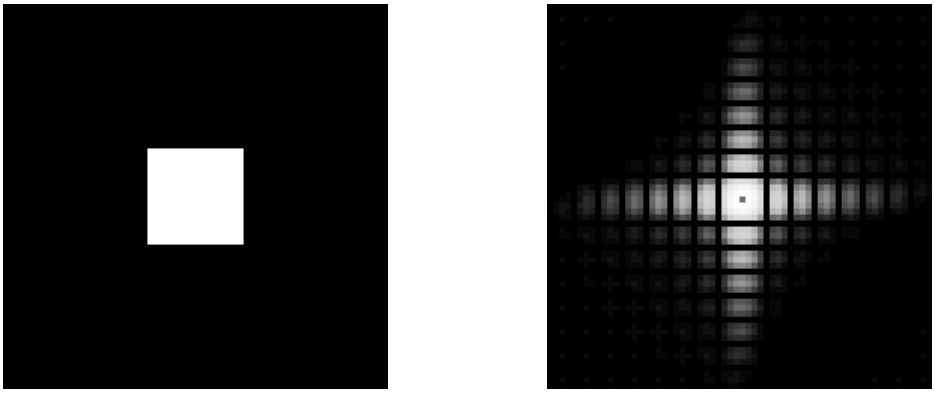

Figure 11. A square image and its 2D FFT.

Figure 12. A more complex image and its 2D FFT.

used in each of the $\log_2(N)$ stages of the calculation may also be precomputed and stored separately for repeated use.

Also, the rearranging of the output data via the bit reversal method can be postponed until after the complete image is processed. Since each row in the image will be ordered in the same manner, the calculation in the second dimension, while performing the computations in a bit reversed order, still performs all of the necessary calculations.

Finally, some mention should be made of the method for transforming the frequency domain version of a signal or image back into the standard digitally sampled representation. While this is not necessary for many applications, others which use the DFT as an intermediate step in processing a signal will require some means of returning the signal to its original representation.

The inverse transform is very similar to the forward transform. In fact, the inverse transform is nothing but the forward transform, with two additional processing steps. These two additional processing steps are the taking of the complex conjugate of each point and dividing the result by N. This procedure returns data in the frequency domain to data in the sampled domain. This inverse transform may be easily implemented by modifying the FFT code.

2.3. Performance

From Figure 8, we see that each stage in the FFT processes vectors of length $N/2$. These vectors are processed for $\log_2(N)$ stages. This gives a total of $(N/2)\log_2(N)$ data elements processed.

Assuming that throughput is maintained such that one data element per cycle is processed, the time to calculate a FFT of length N is given by Equation 6, where f is the clock frequency of the system in Hertz (Hz).

$$T_{fft} = (N/2f) * \log_2(N) \tag{6}$$

From this analysis, the reconfigurable hardware should be able to process a 1024 point FFT in approximately 0.1 milliseconds, assuming a 50 MHz clock. Sohie and Chen[6] give performance benchmarks for modern digital signal processors. These processors typically perform an identical calculation in one to two milliseconds. This predicted order of magnitude increase in performance is especially significant when one considers that the processors executing the FFT benchmarks are not standard microprocessors, but digital signal processors which are optimized for operations such as the FFT.

This order of magnitude increase in performance should allow the two dimensional Fourier transform to be calculated at rates approaching that of standard video. Assuming a video rate of 30 frames per second, a reconfigurable system should able to produce the two dimensional Fourier transform of a $256x256$ pixel video signal in real time while executing at a system clock speed of just over 15 MHz. A $512x512$ signal can be processed in real time if a system clock speed of just of 70 MHz can be achieved. The ability to process video data in this manner should permit filtering and enhancement of video images not normally available using standard techniques.

REFERENCES

1. Steven C. Chapra and Raymond P. Canale. *Numerical Methods for Engineers.* McGraw–Hill Book Company, second edition, 1988.
2. Raphael C. Gonzalez and Paul Wintz. *Digital Image Processing.* Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1987.
3. Steven A. Guccione and Mario J. Gonzalez. A data-parallel programming model for reconfigurable architectures. In Duncan A. Buell and Kenneth L. Pocek, editors, *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87, Los Alamitos, CA, April 1993. IEEE Computer Society Press.
4. Alan V. Oppenheim, Alan S. Willsky, and Ian T. Young. *Signals and Systems.* Prentice–Hall, 1983.
5. William T. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C.* Cambridge University Press, second edition, 1992.
6. Guy R. Sohie and Wei Chen. *Implementation of Fast Fourier Transforms on Motorola's Digital Signal Processors.* Motorola, Inc., 1993.
7. Harold S. Stone. Parallel processing with a perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, February 1971.
8. Earl E. Swartzlander, Jr. and George Hallnor. High speed FFT processor implementation. In *VLSI Signal Processing*, pages 27–34. IEEE Press, 1984.