A General Comparison Of Fft Algorithms

Manish Soni,
Padma Kunthe

Abstract

A large number of FFT algorithms have been developed over the years, notably the Radix-2, Radix-4, Split- Radix, Fast Hartley Transform (FHT),, Quick Fourier Transform (QFT),, and the Decimation-in-Time-Frequency (DITF), algorithms. How these algorithms fare in comparison with each other is of considerable interest to developers of signal processing technology. In this paper, we present a general analysis and comparison of the aforementioned algorithms. The analysis of each algorithm includes the number of mathematical operations, computation time and memory requirements. The results indicate that the FHT is the overall best algorithm on all platforms, offering the fastest execution time and requiring reasonably small amounts of memory.

1. Introduction: The first major breakthrough in implementation of Fast Fourier Transform (FFT), algorithms was the Cooley-Tukey [1] algorithm developed in the mid-1960s, which reduced the complexity of a Discrete Fourier Transform from O(N2), to O(N·logN), At that time, this was a substantial saving for even the simplest of applications. Since then, a large number of FFT algorithms have been developed. The Cooley-Tukey algorithm became known as the Radix-2 algorithm and was shortly followed by the Radix-3, Radix-4, andMixed Radix algorithms [8].
Further research led to the Fast Hartley Transform (FHT), [2,3,4] and the Split Radix (SRFFT), [5] algorithms. Recently, two new algorithms have emerged: the Quick Fourier Transform (QFT), [6] and the Decimation-In-Time-Frequency (DITF), algorithm [7]. In this paper we provide a comparison of several contemporary FFT algorithms. The criteria used are the operations count, memory usage and computation time. We chose the following algorithms for our analysis: Radix-2 (RAD2),, Radix-4 (RAD4),, SRFFT, FHT, QFT and DITF.

2. Review of FFT algorithms: The basic principle behind most Radixbased FFT algorithms is to exploit the symmetry properties of a complex exponential that is the cornerstone of the Discrete Fourier Transform (DFT), These algorithms divide the problem into similar sub-problems (butterfly computations), and achieve a reduction in computational complexity. All Radix algorithms are similar in structure differing only in the core computation of the butterflies. The FHT differs from the other algorithms in that it uses a real kernel, as opposed to the complex exponential kernel used by the Radix algorithms. The QFT postpones the complex arithmetic to the last stage in the computation cycle by separately computing the Discrete Cosine Transform (DCT), and the Discrete Sine Transform (DST), The DITF algorithm uses both the Decimation-In-Time (DIT), and Decimation-In-Frequency (DIF), frameworks for separate parts of the computation to achieve a reduction in the computational complexity.

Radix-2 Decimation in Frequency Algorithm: The RAD2 DIF algorithm is obtained by using the divide-and conquer approach to the DFT problem. The DFT computation is initially split into two summations, one of which involves the sum over the first data points and the other over the next data points, resulting in[11,12].

*Research Scholor M.Tech (Digital Comm.), TIT,College,Bhopal
**Astt.Prof.,Head EC Deptt. TIT,College,Bhopal

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} x(n) . W_N^{kn} + \sum_{n=N/2}^{N-1} x(n) . W_N^{kn} \quad (1)$$

the above equation can be simplified to

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} [x(n) + (-1)^k . x(n + \frac{N}{2})] . W_N^{kn} \quad (2)$$

Considering the even and odd-numbered frequency samples separately results in

$$X(2k) = \sum_{n=0}^{\frac{N}{2}-1} [x(n) + x(n + \frac{N}{2})] . W_{N/2}^{kn} \quad (3)$$

$$X(2k+1) = \sum_{n=0}^{\frac{N}{2}-1} \{(x(n) - x(n + \frac{N}{2})) . W_{N/2}^{kn}\} . W_{N/2}^{kn} \quad (4)$$

The same computational procedure can be repeated through decimation of the N/2- point DFTs X(2k), and X(2k+1), The entire process involves v = log2N stages with each stage involving N/2 butterflies. Thus the RAD2 algorithm involves N/2·log2N complex multiplications and N·log2N complex additions, or a total of 5N·log2N floating point operations. Observe that the output of the whole process is out-of-order and requires a bit reversal operation to place the frequency samples in the correct order.

2.2 Radix-4 Algorithm: The RAD4 algorithm is very similar to the RAD2 algorithm in concept. Instead of dividing the DFT computation into halves as in RAD2, a four-way split is used. The N-point input sequence is split into four subsequences, x(4n),, x(4n+1), , x(4n+2), and x(4n+3), , where n=0,1,…N/4-1. Then,

$$X(k) = \sum_{n=0}^{\frac{N}{4}-1} x(n) . W_N^{kn} + \sum_{n=N/4}^{\frac{N}{2}-1} x(n) . W_N^{kn} + \sum_{n=N/2}^{\frac{3N}{4}-1} x(n) . W_N^{kn} + \sum_{n=3N/4}^{N-1} x(n) . W_N^{kn} \quad (5)$$

Setting

$$F(l, q) = \sum_{m=0}^{\frac{N}{4}-1} x(l, m) . W_{N/4}^{mq} \quad (6)$$

$$X(p,q) = X[\frac{N}{4} \cdot p + q], \quad (7)$$

and,

$X(l,m) = x(4m+l)$, where

l,p=0,1,2,3 & m,q=0,1,...N/4-1. (8)

the matrix formulation of the butterfly becomes

$$\begin{bmatrix} X(0,q) \\ X(1,q) \\ X(2,q) \\ X(3,q) \end{bmatrix} = \begin{bmatrix} W_N^0 & W_N^0 & W_N^0 & W_N^0 \\ W_N^q & -jW_N^q & -W_N^q & jW_N^q \\ W_N^{2q} & -W_N^{2q} & W_N^{2q} & -W_N^{2q} \\ W_N^{3q} & jW_N^{3q} & -W_N^{3q} & -jW_N^{3q} \end{bmatrix} \begin{bmatrix} F(0,q) \\ F(1,q) \\ F(2,q) \\ F(3,q) \end{bmatrix} \quad (9)$$

The decimation process is similar to the RAD2 algorithm, and uses *v*=log4N stages, where each stage has N/4 butterflies. TheRAD4 butterfly involves 8 complex additions and 3 complex multiplications, or a total of 34 floating point operations. Thus, the total number of floating point operations involved in the RAD4 computation of an N-point DFT is 4.25log2N, which is 15% less than the corresponding value for the RAD2 algorithm.

2.3. Split-Radix Algorithm: Standard RAD2 algorithms are based on the synthesis of two half-length DFTs and similarly RAD4 algorithms are based on the fast synthesis of four quarter-length DFTs. The SRFFT algorithm is based on the synthesis of one half-length DFT together with two quarter-length DFTs. This is possible because, in the RAD2 computations, the even-indexed points can be computed independent of the odd indexed points. The SRFFT algorithm uses the RAD4 algorithm to compute the odd numbered points. Hence, the N–point DFT is decomposed into one N/2-point DFT and two N/4 -point DFTs.

$$X(2k) = \sum_{n=0}^{\frac{N}{2}-1}[x(n) + x(n+\tfrac{N}{2})] \cdot W^{4kn} \quad (10)$$

$$X(4k+3) = \sum_{n=0}^{\frac{N}{4}-1}[g(n) + j\, f(n)]W^{3n} \cdot W^{4kn} \quad (11)$$

and,

$$X(4k+1) = \sum_{n=0}^{\frac{N}{4}-1}[g(n) - j\, f(n)]W^{n} \cdot W^{4kn} \quad (12)$$

where,

$$g(n) = x(n) - x(n+\tfrac{N}{2}) \text{ and}$$

$$f(n) = x(n+\tfrac{N}{4}) - x(n+\tfrac{3N}{4}) \quad (13)$$

An N-point DFT is obtained by successive use of these decompositions. Here we treat the computational process as a RAD2 algorithm with the unnecessary intermediate DFT computations eliminated. An analysis of the butterfly structures [15] for the SRFFT algorithm reveals that approximately 4N·log2N computations are required as compared to 4.25log2N for RAD4 and 5N·log2N for RAD2 algorithms.

2.4. Fast Hartley Transform: The main difference between the DFT computations previously discussed and theDiscrete Hartley Transform (DHT), is the core kernel [2,14]. For the DHT, the kernel is real unlike the complex exponential kernel of the DFT. The DHT coefficient is expressed in terms of the input data points as[13]

$$X(k) = \sum_{n=1}^{N-1} x(n) \cdot [\cos(\tfrac{2\pi nk}{N}) + \sin(\tfrac{2\pi nk}{N})] \quad (14)$$

This results in the replacement of complex multiplications in a DFT by real multiplications in a DHT. For complex data, each complex multiplication in the summation requires four real

multiplications and two real additions using the DFT. For the DHT, this computation involves only two real multiplications and one real addition. There exists an inexpensive mapping of coefficients from the Hartley domain to the Fourier domain, which is required to convert the output of a DHT to the traditional DFT coefficients. Equation (15), relates the DFT coefficients to the DHT coefficients for an N-point DFT computation.

$$Re\big(DFT(k)\big) = \frac{DHT\,(k) + DHT\,(N-k)}{2}$$
$$Im\big(DFT(k)\big) = \frac{DHT\,(k) - DHT\,(N-k)}{2} \qquad (15)$$

The FHT evolved from principles similar to those used in the RAD2 algorithm to compute DHT coefficients efficiently[13]. It is intuitively simpler and faster than the FFT algorithms as the number of computations reduces drastically when we replace all complex computations by real computations. Similar to other recursive Radix algorithms, the next higher order FHT can be obtained by combining two identical preceding lower order FHTs. In fact all Radix-based algorithms used in FFT implementations can be applied to FHT computations [16]. For N=2, the Hartley transform can be represented in a matrix form as

$$\begin{bmatrix} X(0) \\ X(1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \end{bmatrix} \qquad (16)$$

Following a similar procedure for, we get the matrix formulation,

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix} \qquad (17)$$

which can be easily transformed to

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} x(0) \\ x(2) \\ x(1) \\ x(3) \end{bmatrix} \qquad (18)$$

A closer look at this matrix product and a comparison with the matrix for N=2 reveals that the matrix for N=4 is composed of sub-matrices of the form of the matrix for N=2. Thus a DHT of order 4 can be computed directly from a DHT of order 2. This idea can be extended to any order which is a power of 2 [4]. It is also worth noting that the Hartley Transform is a bilateral transform, i.e. the same functional form can be used for both the forward and inverse transforms. This is an added advantage of the FHT over other FFT algorithms.

2.5. Quick Fourier Transform: We have seen that the Radix-based algorithms exploit the periodic properties of the cosine and sine functions. In the Quick Fourier Transform (QFT), algorithm, the symmetry properties of these functions are used to derive an efficient algorithm.

$$\cos\left(\frac{2\pi(N-n)k}{N}\right) = \cos\left(\frac{2\pi nk}{N}\right)$$
$$\sin\left(\frac{2\pi(N-n)k}{N}\right) = -\sin\left(\frac{2\pi nk}{N}\right) \quad (19)$$

We define an N+1 -point DCT as

$$X_{DCT}(k) = \sum_{n=0}^{N} x(n)\cos\frac{\pi nk}{N}, \quad k = 0,1,\ldots,N \quad (20)$$

An N-1-point DST can also be similarly defined as

$$X_{DST}(k) = \sum_{n=1}^{N-1} x(n)\sin\frac{\pi nk}{N}, \quad k = 0,1,\ldots,N \quad (21)$$

We can divide an N-point input sequence into its even and odd parts as

$$x_e(0) = x(0)$$
$$x_e(k) = x(k) + x(N-k),\ k = 1, 2,\ldots,N/2 - 1 \text{ and}$$
$$x_e(N/2) = x(N/2) \quad (22)$$

$$x_0(k) = x(k) - x(N-k),\ k = 1,2,\ldots,N/2 - 1 \quad (23)$$

Using the above sequences and properties in Equation 19 we can define an N-point DFT as

$$X(k) = X_{DCT}(k) - j\,X_{DST}(k),$$

$$X(N-k) = X_{DCT}(k) + jX_{DST}(k),\ k=1,2\ldots,N/2-1 \quad (24)$$

In order to derive a recursive formulation of DCT and DST computations, we define a new sequence, *xe* as

$$x_e(k) = x(k) + x(N-k),\ k = 1,2\ldots,N/2 - 1 \quad (25)$$

Also, the *N/*2th point of this sequence is the same as that of the original sequence. Thus we can formulate the recursive DCT for the even numbered points as

$$DCT(2k, N+1, x) = DCT\left(k, \tfrac{N}{2}+1, x_e\right)$$
$$\text{where } k = 1,2\ldots,N/2 - 1 \quad (26)$$

$$X(0) = X_{DCT}(0) \text{ and } X(N/2) = X_{DCT}(N/2) \quad (27)$$

We can define a recursive equation for the odd DCT points using a new sequence *xo* defined as

$$x_o(k) = \frac{x(k)-x(N-k)}{2\cos(\pi k/N)},\ k=1,2\ldots,N/2-1 \quad (28)$$

Then,

$$DCT(2k+1, N+1, x) = DCT\left(k, \tfrac{N}{2}+1, x_o\right) +$$
$$DCT\left(k+1, \tfrac{N}{2}+1, x_o\right), \quad k = 0,1,\ldots,N \quad (29)$$

A similar recursive formulation can be derived for the DST using symmetry properties of the sine function which results in

$$DST(2k, N-1, x) = DST\left(k, \frac{N}{2}\text{-}1, x_o\right) \text{ and}$$

$$DST(2k+1, N-1, x) =$$

$$DST\left(k, \frac{N}{2}\text{-}1, x_e\right) + DST\left(k+1, \frac{N}{2}\text{-}1, x_e\right) \qquad (30)$$

where k =1,2…,N/2–1.Since the complex operations occur only in the last stage of the computation where the DCT and DST are combined using Equation 24, the QFT is well suited for operation on real data. The number of operations required to perform an N- point QFT is $11N/2 \cdot \log N - 27N/4+2$ [7].This, however, does not include the cost of computing the odd and even parts of the data sequence at each stage of the computation.

2.6. Decimation-In-Time-Frequency (DITF), Algorithm: The DITF algorithm is based on the observation that in a DIF implementation of a RAD2 algorithm, most of the computations (especially complex multiplications), are performed during the initial stages of the algorithm. In the DIT implementation of the RAD2 algorithm, the computations are concentrated towards the final stages of the algorithm. Thus, starting with the DIT implementation and then shifting to the DIF implementation at some transition stage intuitively seems to be a computation saving process. Equations (3), and (4), define the DIF RAD2 computation. The DIT RAD2 computation is defined as

$$X(k) = \sum_{n=0}^{\frac{N}{2}-1} [x(2n).W_{\frac{N}{2}}^{kn} + \left[\sum_{n=0}^{\frac{N}{2}-1} x(2n+1).W_{\frac{N}{2}}^{kn}\right].W_N^k \qquad (31)$$

Note that the first summation in the above equation is the N/2-point DFT of the sequence comprised of the even-numbered points of the original sequence and the second summation is the N/2-point DFT of the sequence comprised of the odd numbered points of the original sequence. The transition stage consists of a conversion from the DIT coefficients to the DIF coefficients,

$$DIF(k) = W_N^{pq} \cdot DIT(k) \qquad (32)$$

where p is the index of the set to which k belongs and q is the position of k in that set. The indices of each set need to be bit reversed. The total number of real multiplications involved in the DITF computation is $2N \cdot \log N - 10N + 8N/2s + 8.2s - 8$, where s is the transition stage. On minimizing this expression, we get the optimal transition stage for minimum number of multiplications as

$$\frac{\log N}{2}.$$

3. Benchmarking criteria: Most preceding FFT complexity studies have been conducted on special purpose hardware such as digital signal processing (DSP), chips [9,10]. Typically, the primary benchmarking criteria have been the number of mathematical operations (multiplications and additions), and/or the overall computation speed. The efficiency of an algorithm is most influenced by the arithmetic complexity, usually expressed in terms of a count of real multiplications and additions. However, on general purpose computers this is not a very good

benchmark and other factors need to be considered as well. For instance, the issue of memory usage is very important for memory constrained applications.

3.1. Number of Computations: Since many general purposes CPUs have significantly different speeds on floating point and integer operations, we decided to individually account for floating point and integer arithmetic. It is a well known fact that most new architectures compute floating point operations more efficiently than integer operations [19,21]. Also, most indexing and loop control is done using integer arithmetic. Therefore the integer operations count directly measures the cost of indexing and loop control. Many FFT algorithms require a large number of division-by-two operations which is efficiently accomplished by using a binary shift operator. To account for this common operation, we include a count of binary shifts in our benchmarks.

3.2. Computation Speed: In most present-day applications for general purpose computers, with easy availability of faster CPUs and memory not being a primary constraint, the fastest algorithm is by far treated as the best algorithm. Thus, a common choice to rank order algorithms is by their computation speed.

3.3. Memory Usage: One of the classic trade-offs seen in algorithm development is that of memory usage versus speed. In most portable signal processing applications, the FFT is a core computational component. However, few applications can afford a large memory space for evaluating FFTs. While memory usage is important for specification of hardware, memory accesses also account for a significant portion of computation time. This is attributed to cache misses, swapping and other paging effects. These effects are more prominent when computing higher order FFTs (typically over 4K points),These observations prompted us to include memory usage as one of the yardsticks in judging the effectiveness of the various FFT algorithms.

4. Benchmarking results and analysis: Each of the algorithms was implemented under a common framework using common functions for operations such as bit-reversal and lookup table generation so that differences in performance could be attributed solely to the efficiency of the algorithms. Following this, we comprehensively benchmarked each algorithm according to the criteria discussed in the previous section.

4.1. Computation Speed: Computation speed is typically the most prominent aspect of an FFT algorithm in current DSP applications. The computation speed of an algorithm for large data sizes can often be heavily dependent on the clock speed, RAM size, cache size and the operating system. Hence, these factors must be taken into account. We evaluated that the computation time of the worst algorithm (DITF), is more than three times greater than that of the best algorithm (FHT), It has been consistently observed in our benchmarks that the FHT is the most efficient algorithm in terms of computation speed. Table 1 shows the variation in performance of these algorithms as a function of the FFT order. The performance is clearly affected by the amount of RAM and cache. As expected, the effect is more pronounced for higher order FFTs where cache misses become common. The performance of the CPUs on floating point operations versus integer operations is significant as well.

4.2. Number of Computations: The number of arithmetic computations has been the traditional measure of algorithmic efficiency. The numbers of operations required by each algorithm for a 1024-point real DFT are displayed in Table 2. We observe that the faster algorithms require performing a smaller number of computations. However, there is a trade-off between integer operations and floating point operations. Savings in floating point operations can be achieved at the cost of increasing the number of integer operations. An example of this is seen in the excessive number of integer additions in the QFT. In the QFT implementation, the DCT and DST

recursions are implemented by accessing pointers in a common workspace. This results in the large number of integer operations. The large numbers of operations for the DITF algorithm are attributed to the bit-reversal process at various stages of the computation. This aspect seems to have been overlooked in previous evaluations [7,8]. Overall, the FHT and the SRFFT are the best in terms of effectively using computations, which translates to greater computation speed. The main drawback of the FHT is that the complex FHT is computed via two real FHT computations. The QFT also uses a similar methodology. The number of computations doubles when moving from real data to complex data using these algorithms. The corresponding change for the other algorithms is insignificant.

4.3. Memory Usage: One of the key issues in portable applications is memory usage. Table 3 shows the memory usage profile of different algorithms for a 1024 point FFT. We see from Table 3 that the RAD2 algorithm is the most memory efficient algorithm, and the QFT is the least. In the case of the QFT, this is due to the large work space required to perform the recursions in the DCT and the DST algorithms. The FHT is the most inefficient in terms of the executable size. Notice that, as was expected, the executable size is a good measure of the complexity of the algorithm with the FHT being the most complex and the RAD2 the least complex algorithm.

5. Conclusions: The existence of an abundance of algorithms for FFT computations and an even greater number of their implementations calls for a comprehensive benchmark which teases out the implementation-specific differences and compares the algorithms directly. We have tried to achieve this objective by implementing algorithms in a very consistent framework. Our results indicate that the overall best algorithm for DFT computations is the FHT algorithm. This has been, and will likely continue to be, a point of argument for many years [17, 18, 20, 22, 23]. Another feature in favor of the FHT is its bilateral formulation. UnlikeDFT algorithms, FHT has the same functional form for both its forward and inverse transforms. The FHT is the fastest algorithm on all platforms with a reasonable dynamic memory requirement. However, it is the most inefficient in terms of static memory usage (measured in terms of the executable size), If an FFT algorithm needs to be chosen solely on the basis of static memory requirements, the RAD-2 algorithm is the still the best, owing to its simple implementation. The SRFFT and the FHT are comparable in terms of the number of computations and are the most efficient.

Algorithm	**FFT order**					
	16	**64**	**256**	**1024**	**4096**	**16384**
RAD2	20	60	260	1960	6800	30500
RAD4	20	60	300	1800	6940	29000
SRFFT	20	40	140	660	3700	17260
FHT	**20**	**40**	**120**	**560**	**3240**	**14020**
QFT	20	40	180	1020	5460	27760
DITF	20	80	380	1780	8780	40200

Table 1: Computation time (in microseconds),of various algorithms.

Algorithm	Float Adds	Float Mults	Integer Adds	Integr Mults	Binary Shifs
RAD2	14336	20480	19450	2084	1023
RAD4	8960	14336	12902	3071	277
SRFFT	5861	5522	12664	2542	1988
FHT	**7420**	**8841**	**3235**	**2048**	**12**
QFT	9026	2560	29784	1048	144
DITF	14400	17664	20333	1076	1074

Table 2: Number of computations involved in computing a 1024-point FFT.

Algorithm	Memory Usage (Bytes)
RAD2	**72240**
RAD4	72536
SRFFT	72508
FHT	72652
QFT	122072
DITF	78632

Table 3: Memory usage in computing a 1024-point FFT.

References:

1. J.W. Cooley and J.W. Tukey, (1965), An Algorithm for Machine Computation of Complex Fourier Series, Mathematical Computation, vol. 19, pp. 297-301.
2. R.N. Bracewell, (1985), The Hartley Transform, Oxford Press, Oxford, England.
3. R.N. Bracewell, (1984), Fast Hartley Transform, Proceedings of IEEE, pp. 1010-1018.
4. H.S. Hou, (1987), The Fast Hartley Transform Algorithm, IEEE Transactions on Computers, pp. 147-155, February.
5. P. Duhamel and H. Hollomann, (1984), Split Radix FFT Algorithm, Electronic Letters, vol. 20, pp. 14-16, January.
6. H. Guo, G.A. Sitton, and C.S. Burrus, (1994), The Quick Discrete Fourier Transform, Proceedings of International Conference on Acoustics, Speech and Signal Processing, vol. 3, pp. 445-447, Adelaide, Australia.
7. A. Saidi, (1994), Decimation-In-Time Frequency FFT Algorithm, Proceedings of International Conference on Acoustics, Speech and Signal Processing, vol. 3, pp. 453-456, Adelaide, Australia.
8. C.S. Burrus and T.W. Parks, (1985), DFT/FFT and Convolution Algorithms: Theory and Implementation, John Wiley and Sons, New York, NY, USA.
9. http://www.ti.com/sc/docs/dsps/literatu.htm
10. http://www.lsidsp.com/c6x/tech/wpsy nop.htm
11. J.G. Proakis, D.G. Manolakis, (1992), Digital Signal Processing - Principles, Algorithms and Applications, Macmillan Publishing Company, NY, USA, 1992.
12. A.V. Oppenheim, R.W. Schafer, (1989), Digital Signal Processing, Prentice-Hall International Inc., Englewood Cliffs, NJ, USA.
13. R.N. Bracewell, (1990), Assessing the Hartley Transform, IEEE Transactions on Acoustics Speech and Signal Processing, pp. 2174-2176.
14. M. Popovic, D. Sevic, (1994), A new look at the Comparison of Fast Hartley and Fourier Transforms, IEEE Transactions on Signal Processing, vol. 42, pp. 2178 2182.
15. C.V. Loan, (1992), Frontiers in Applied Mathematics - Computational Frameworks for the Fast Fourier Transform, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
16. H.V. Sorensen, D.L. Jones, M.T. Hiedeman, and C.S. Burrus, (1985), On Computing the Discrete Hartley Transform, IEEE Transactions on Acoustics Speech and Signal Processing, pp. 1231-1238.
17. M. Popovic, D. Sevic, A new look at the Comparison of Fast Hartley and Fourier Transforms,
18. P.R. Uniyal, (1994), Transforming Real-Valued Sequences: Fast Fourier versus Fast Hartley Transform Algorithms, IEEE Transactions on Signal Processing, vol.42, pp. 3249-3253.
19. http://www.contrib.andrew.cmu.edu/usr/sdavis/processor.html
20. P. Duhamel and M. Vetterli, (1987), Improved Fourier and Hartley Transform Algorithms: Application to Cyclic Convolution of Real Data, IEEE Transactions on Acoustics Speech and Signal Processing, pp. 818-824.
21. http://www.digital.com/semiconductor/micro-rpt-21164.htm.
22. H.V. Sorensen, D.L. Jones, M.T.Hiedeman, and C.S. Burrus, (1985), On Computing the Discrete Hartley Transform, IEEE Transactions on Acoustics Speech and Signal Processing, pp. 1231-1238.
23. R.D. Preuss, Very Fast Computation of the Radix-2, Discrete Fourier Transform, IEEE Transactions on Acoustics Speech and Signal Processing,