Research Article

Engineering: Open Access

A Comparative Analysis of CORDIC-Based versus LUT+Multiplier-Based Schemes for Parallel Computation of Radix-4 FFT Dragonfly

Dr. Keith Jones*

Consultant Mathematician, Retired, Weymouth, Dorset, UK

***Corresponding Author**
Keith Jones, Consultant Mathematician, Retired, Weymouth, Dorset, UK.

Submitted: 2026, June 15; **Accepted**: 2026, July 12; **Published**: 2026 July 17

Citation: Jones, K (2026). A Comparative Analysis of CORDIC-Based Versus LUT+Multiplier-Based Schemes for Parallel Computation of Radix-4 FFT Dragonfly. *Eng OA, 4*(7), 01-16.

Abstract

This paper analyzes the relative complexities, as expressed in terms of arithmetic and trigonometric memory requirements, of three different solutions for the parallel computation of those arithmetic tasks needing to be performed by the computational unit of the radix-4 fast Fourier transform (FFT) algorithm, denoted CU_4 and referred to here as a 'dragonfly'. The first two solutions are conventional, being based upon different combinations of look-up tables (LUTs), adders and multipliers: the first solution, employing single-level LUTs combined with adders and multipliers, seeks to minimize the arithmetic complexity at the expense of an increased memory requirement, whilst the second, employing a two-level LUT and second-order recursion (involving just adders and multipliers) combined with additional adders and multipliers, seeks to minimize the memory requirement at the expense of increased arithmetic complexity. The third solution takes a non-conventional approach, being based upon the use of three pipelined COordinate Rotation DIgital Computer (CORDIC) arithmetic units (each employing its own adders, together with bit shifters and constant multipliers) combined with additional adders, and has a complexity, unlike that of the other two solutions, that's dependent upon the word length but independent of the transform length, effectively eliminating the trigonometric memory requirement at the expense of an increased number of adders. With the transform length and word length denoted by N and L, respectively, these properties follow directly from the derived complexity results which state that the first solution possesses $O(LN)$ complexity, the second $O(L\sqrt{N})$ complexity and the third $O(L^2)$ complexity, where N >> L, with the results generalizing in an obvious fashion to the case of CU_R, for the radix-R FFT, where R is an arbitrary positive integer. As a result, for those big-data applications involving the use of large transforms, independence of the transform length makes the CORDIC-based solution look particularly attractive, especially when resources – such as fast memory – are limited.

Keywords: Complexity, CORDIC, Dragonfly, FFT, LUT, Parallelism

1. Introduction

The aim of this paper is to provide an analysis of the relative *complexities*, as expressed in terms of arithmetic and trigonometric memory requirements, of three different solutions for the parallel computation of those arithmetic tasks needing to be performed by the fixed-radix fast Fourier transform (FFT) algorithm's *computational unit*, denoted CU_R for the radix-R version of the algorithm [1,2]. The FFT refers to a class of algorithms that provide a fast means of computing the discrete Fourier transform (DFT) algorithm, with CU_R being the computational engine used for carrying out the radix-R algorithm's repetitive arithmetic operations resulting from its recursive structure [3]. The DFT, for the case of a length-N transform, may be expressed in normalized form via the equation

$$X[k] = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x[n].W_N^{nk} \qquad k = 0, 1, \ldots, N-1 \qquad (1)$$

where the input/output data sets belong to C^N, the linear space of complex-valued N tuples, and where the transform kernel is expressed in terms of powers of W_N, where

$$W_N = \exp(-i2\pi / N), \qquad i = \sqrt{-1}, \qquad (2)$$

the primitive N^{th} complex root of unity [4]. The complex-valued exponential terms, W_N^{nk}, as derived from the complex root of unity, each comprise two trigonometric components, one *sinusoidal* and the other *cosinusoidal*, with each pair being more commonly referred to as *twiddle factors* [1,2]. These terms, for the case of a radix-R transform of length N, whereby

$$N = R^E \tag{3}$$

for some positive integer exponent E, are required to be fed, R at a time, to CU_R, although only R-1 of the twiddle factors are *non-trivial* as the first twiddle factor is *trivial* with a fixed value of one. Given the recursive nature of the fixed-radix FFT, it is clear that knowledge of the complexity of CU_R, as is sought here, leads directly to the complexity of the associated FFT algorithm, as the radix-R FFT can be shown to involve the execution of CU_R a total of $(N/R)\times\log_R N$ times – that is, the algorithm comprises $\log_R N$ *temporal stages* where the processing for a given stage cannot commence until that of its predecessor has been completed and where each stage involves N/R instances of CU_R [5].

The complexity of CU_R is to be measured in terms of: 1) the *arithmetic* component, as expressed in terms of the required numbers of adders and multipliers, and 2) the *trigonometric memory* component, which concerns the storage of a limited range of a sampled sine (or, equivalently, cosine) function, via the construction and maintenance of one or more suitably defined *look-up tables* (LUTs), from which the real and imaginary components of the twiddle factors – as required for the operation of CU_R – may be derived. The first two solutions are *conventional*, being based upon recently published results which involve the use of different combinations of LUTs, adders and multipliers [6,7]. The third solution takes a *non-conventional* approach, being based upon the use of multiple *pipelined* COordinate Rotation DIgital Computer (CORDIC) arithmetic units – with each unit involving the use of its own adders, together with bit-shifters and constant multipliers (that is, where each multiplier possesses one fixed multiplicand) – combined with additional adders [8-10]. This solution effectively eliminates the reliance on LUTs which, for those *big data* applications involving the use of very-long (of $O(10^6)$, say, as might be encountered with processing of wideband signals embedded in astronomical data) to ultra-long (of $O(10^9)$, say, as might be encountered with processing of ultra-wideband signals embedded in cosmic microwave data) transforms, might otherwise prove prohibitively costly in terms of *fast on-chip memory*.

The assessment of the three solutions is to be based upon a highly-parallel implementation of a single CU_R, where R is equal to four, denoted CU_4 and chosen for ease of analysis and illustration, this including the associated trigonometric memory requirement. The chosen computing device is to be based upon silicon-based technology as typified by the field programmable gate array (FPGA) [11]. The dominant operator used by CU_R in the hardware implementation of most conventional FFT solutions is that of the complex multiplier (or, more commonly, multiplier-accumulator), which may be performed in one of two ways, using: 1) four real multipliers, with these being followed by two real adders, or 2) three real multipliers, with these being preceded by two real adders and followed by three real adders. Given the emphasis on minimizing the amount of silicon resources required for such an implementation, and that the orders of complexity for the multiplier and adder are given by $O(L^2)$ and $O(L)$, respectively, where L is the word length, the second three-multiplier version is to be the solution of choice.

Thus, following this introductory section, the paper continues in Section 2 with a brief discussion of the complexity of the computational building blocks required by those arithmetic tasks needing to be performed by CU_4. This is followed in Section 3 by an account of the two conventional solutions based upon the use of different combinations of LUTs, adders and multipliers – noting that the adoption of second-order recursion by one of the solutions also depends upon the use of just adders and multipliers. Section 4 then derives the basic CORDIC algorithm as required for the specific operating mode of *phase rotation*, as is of interest here, whilst Section 5 uses the CORDIC approach to derive an alternative non-conventional solution to the parallel computation of CU_4 based upon the use of three pipelined CORDIC arithmetic units. This is followed in Section 6 with a comparison of the relative complexities of all three solutions for various combinations of transform length and word length, with a summary and conclusions provided in Section 7.

2. Complexity of Computational Building Blocks – Arithmetic and Memory

The main problem to be addressed in this paper is concerned with the *generation and application* of the non-trivial twiddle factors by CU_4. In assessing the associated complexity it is first proposed that L-bit fixed-point processing and a transform length of N should be adopted, where $N >> L$. Also, for ease of analysis and illustration, it is proposed that a somewhat simplistic measure of the required resources be adopted whereby the size of each arithmetic operator and storage location is expressed in terms of the required numbers of *programmable logic slices* – although the adoption of *embedded functions*, as provided by the device manufacturer, would be expected to provide more highly optimized results. This is achieved by making the following assumptions:- that 1) each real adder requires L/2 slices; 2) each real multiplier requires a (worst-case) figure of L^2 slices; so that as a consequence, 3) each complex multiplier, expressed as the combination of three real multipliers and five real adders, requires a (worst case) figure of $3L^2+5L/2$ slices; 4) each constant real multiplier requires L slices; 5) each bit-shifter, assumed *hard-wired* when shift lengths may be predetermined (as is the case for the pipelined CORDIC arithmetic unit to be discussed in Sections 4 and 5), requires negligible resources when compared to that of a

conventional barrel shifter; and 6) each word of trigonometric memory (assumed here to be of *dual-port* type for the simultaneous access of both sine and cosine components) requires L slices [11].

Note that the *data memory*, as required for the storage of the length-N complex-valued data sets, will be the same (aside from any *double-buffering* requirement for the simultaneous construction and processing of data sets) for all of the solutions considered whilst the control logic requirement will be assumed, for ease of analysis, to be of comparable complexity for all such solutions. Therefore, their contributions, as with the hard-wired bit-shifters, will not be included in the complexity results discussed in this paper – although the contribution of the data memory requirement will be crucial in assessing the complexity and viability of different solutions (and, in particular, of their computing architectures) to the associated FFT, particularly for big-data applications.

In assessing the relative complexities of the three solutions discussed here, for various combinations of transform length and word length, a parallel solution is to be assumed, whether that parallelism is achieved in the spatial domain, through the adoption of *single-instruction multiple-data* (SIMD) type processing [12], or the *temporal domain*, through the adoption of *pipelined* processing [12], or through a combination of the two. Each arithmetic *operation* is to be assigned its own hardware *operator* and each CU_4 (or set of CU_4's) its own *processing element* (PE) containing the required hardware operators, with all the twiddle factors being produced and made available for input to CU_4 simultaneously, together with those samples of complex-valued data with which the twiddle factors are to be appropriately combined. Also, although the arithmetic operators and trigonometric memory are expressed in terms of real operators and real words, respectively, the arithmetic is assumed to be of fixed-point type so that the operators are actually assumed to be performing upon L-bit integers.

3. Summary of Recent Results Obtained for Fixed-Radix FFT Algorithm

For the most commonly adopted FFT radices, namely those with values of *two* or *four*, CU_R is more commonly known as a *butterfly* or *dragonfly*, respectively, this naming being due to the fact that the data within the resulting computational structures flow in a pattern that resembles that of the wings of the corresponding insect, respectively [1,2,5]. The structure of CU_R, in each case, is regular and relatively simple, so that the choice of the computationally-efficient radix-4 FFT dragonfly, CU_4, as the preferred algorithm seems a sensible one to make – see Figures 1 and 2 which depict the dragonfly's signal flow graph for the *decimation-in-time* (DIT) and *decimation-in-frequency* (DIF) versions of the radix-4 algorithm, with digit-reversed (DR) input/output addresses being adopted, respectively [1,2]. Generalization of the complexity results follows in an obvious fashion to the case of CU_R, for the radix-R FFT, where R is an arbitrary positive integer. The PE corresponds, therefore, to the hardware required for a highly parallel implementation of a single dragonfly, this including the trigonometric memory requirement needed for its operation – noting that with FPGA technology, the memory is typically available in the form of random access memory (RAM) which, for fast access, needs to be placed on the same device as is used for the processing where it is typically referred to as 'fast RAM' [11].

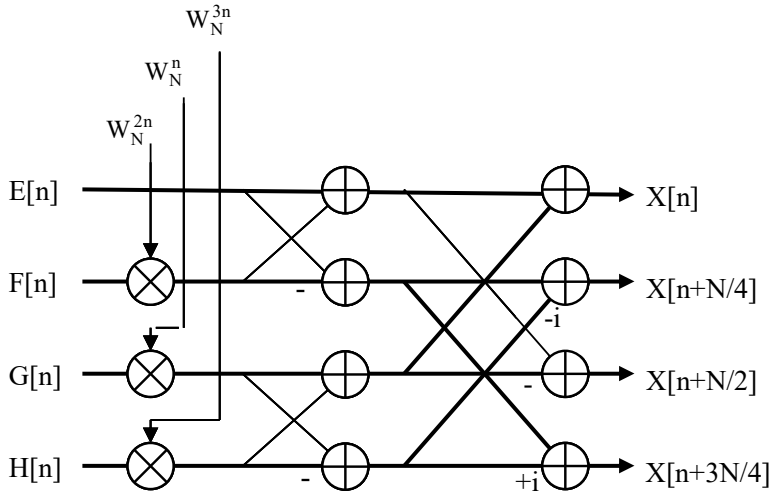

Figure 1: DIT version of radix-4 FFT dragonfly with DR-reordered inputs

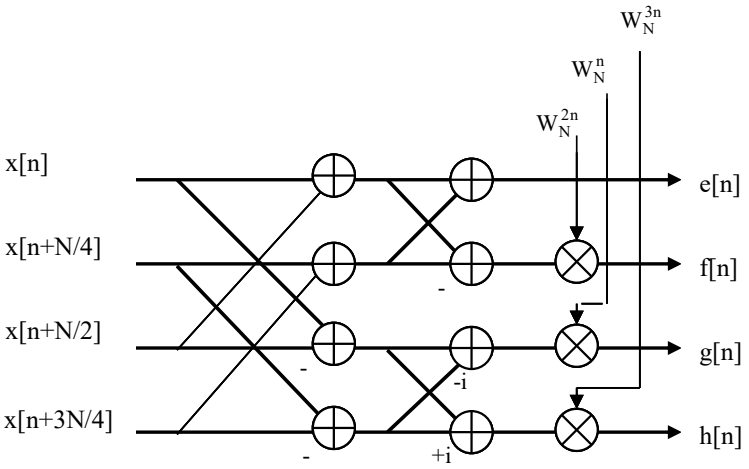

Figure 2: DIF version of radix-4 FFT dragonfly with DR-reordered outputs

3.1 Scheme for Minimizing Dragonfly's Arithmetic Complexity

The task of this first solution, referred to hereafter as **Solution I**, which is based upon recently published results for efficient twiddle factor generation, involves the combined task of the parallel generation and application of the trigonometric components of the three non-trivial twiddle factors, using the *single quadrant scheme* which covers an angular region of 0 to $\pi/2$ radians, combined with adders and multipliers [7]. This approach, for the case of a length-N transform, involves the assigning of a *one-level LUT*, of length N/4, to each non-trivial twiddle factor, enabling all three LUTs to be accessed in parallel in the spatial domain in SIMD type fashion, as illustrated in Figure 3, to facilitate the parallel operation of the PE at *minimal cost in terms of arithmetic complexity*. Thus, the generation component of the solution – which is ideally suited to most applications which require the use of short to long (that is, up to $O(10^3)$, say) transforms – involves an additive complexity of

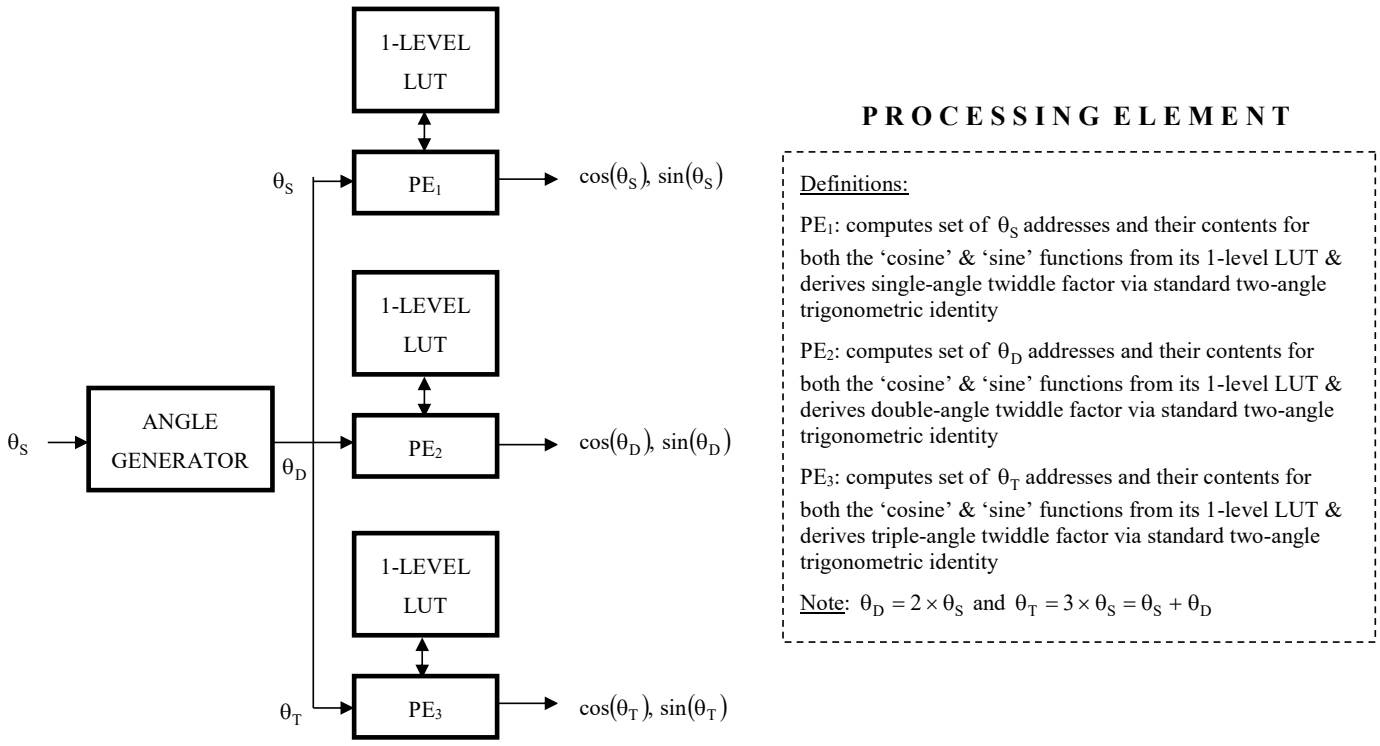

Figure 3: SIMD architecture for generation of dragonfly twiddle factors using single-level LUTs

$$C_{ADDS}^{(1,G)} = 7 \tag{4}$$

real adders, this catering for the address updating of the four complex-valued samples stored within the data memory and the three trigonometric component pairs stored with the LUTs, which are to be appropriately combined, together with an associated trigonometric memory requirement of

$$C_{STORE}^{(1,G)} = 3 \times N/4 \tag{5}$$

words of fast RAM for the construction of the three LUTs.

With regard to the application component of the solution, once the complex-valued data samples and the trigonometric component pairs of the three non-trivial twiddle factors have been generated, it simply remains for them to be appropriately combined. This involves a multiplicative complexity of

$$C_{MULTS}^{(1,A)} = 3 \times 3 = 9 \tag{6}$$

real multipliers, together with an associated additive complexity of

$$C_{ADDS}^{(1,A)} = 3 \times 5 + 16 = 31 \tag{7}$$

real adders, this including 16 adders for carrying out the dragonfly's remaining two stages of pre/post (according to choice of decimation scheme) additions.

As a result, the total complexity of the dragonfly for this first solution, which includes the combined task of both generating and applying the twiddle factors, may be expressed as the combination of the arithmetic component and the trigonometric memory component, where the arithmetic complexity is expressed as the sum of the multiplicative and additive complexities. Thus,

$$C_{MULTS}^{(1)} = C_{MULTS}^{(1,A)} = 9 \tag{8}$$

real multipliers, and

$$C_{ADDS}^{(1)} = C_{ADDS}^{(1,G)} + C_{ADDS}^{(1,A)} = 38 \tag{9}$$

real adders, and

$$C_{STORE}^{(1)} = C_{STORE}^{(1,G)} = 3 \times N/4 \tag{10}$$

words of fast RAM.

3.2 Scheme for Minimizing Dragonfly's Trigonometric Memory Requirement

The task of this second solution, referred to hereafter as **Solution II**, which like the first solution is based upon recently published results [7] for efficient twiddle factor generation, involves the combined task of the parallel generation and application of the trigonometric components of the three non-trivial twiddle factors using a *two-level LUT and second-order recursion* combined with additional adders and multipliers (that is, apart from the recursion's own adders and multipliers). This approach, for the case of a length-N transform, involves the use of a two-level LUT comprising three single-level LUTs, each of length $\sqrt{N/4}$ – one defined over a *coarse-resolution* region and two (one for each of the sine and cosine components) defined over a *fine-resolution* region – for the generation of the first non trivial twiddle factor, and second-order recursion for the generation of the second and third non-trivial twiddle factors. The computation may be carried out in parallel fashion, in the *temporal domain*, via the use of a suitably defined computational pipeline, as illustrated in Figure 4, at *minimal cost in terms of memory*.

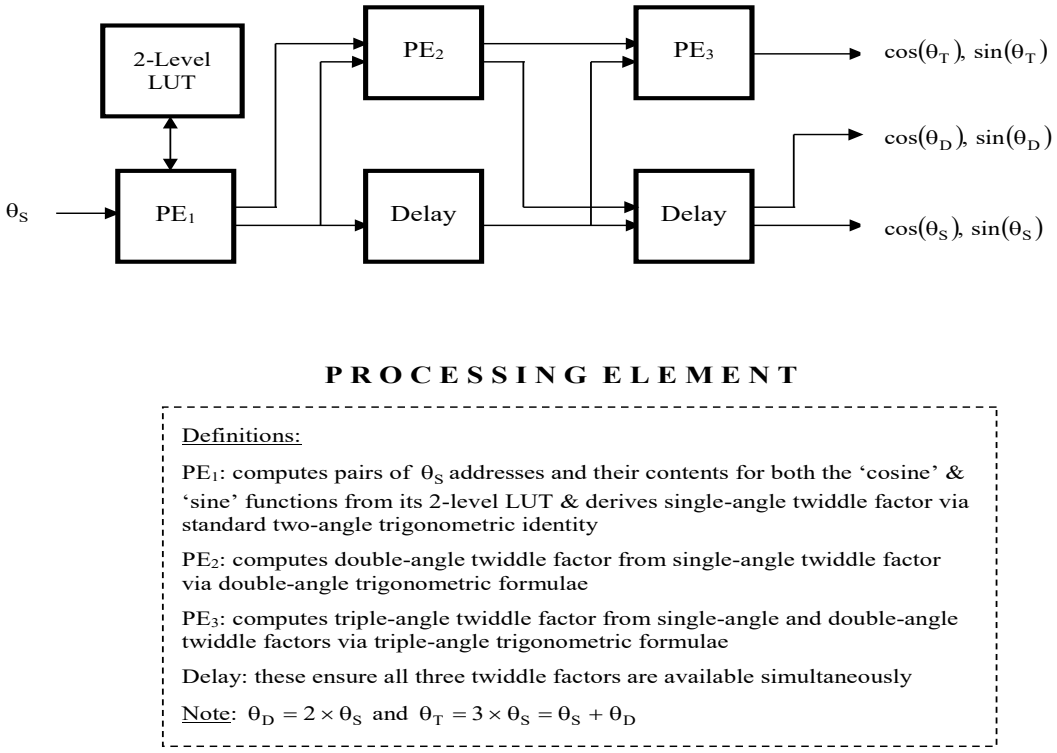

Figure 4: Pipeline architecture for generation of dragonfly twiddle factors using two-level LUT and second-order recursion

For the first non-trivial twiddle factor, the trigonometric formulae required for the addressing of the two-level LUT may be derived in a straightforward manner from the standard two-angle identities

$$\sin(\theta+\phi)=\sin(\theta)\times\cos(\phi)+\cos(\theta)\times\sin(\phi) \tag{11}$$

$$\cos(\theta+\phi)=\cos(\theta)\times\cos(\phi)-\sin(\theta)\times\sin(\phi) \tag{12}$$

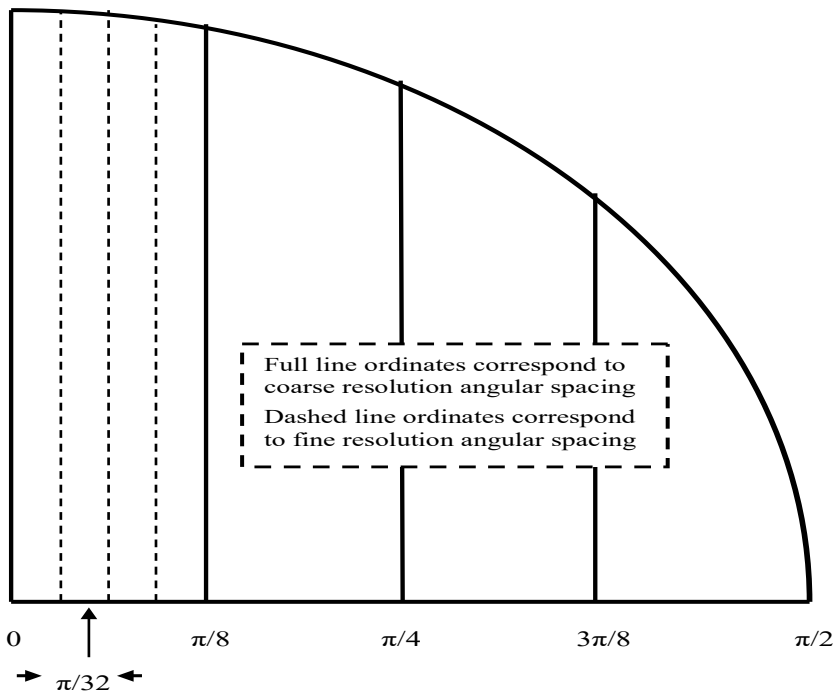

Figure 5: Decomposition of single quadrant of cosine function into coarse-resolution & fine-resolution angular regions using two single-level length-4 LUTs

where ‘θ’ may be viewed as corresponding to the angle defined over a *coarse-resolution* angular region and ‘ϕ’ to the angle defined over a *fine-resolution* angular region – see the simplified illustration of Figure 5 for the decomposition of the cosine function into coarse-resolution and fine-resolution angular regions, each of length four. For the second and third non-trivial twiddle factors, the validity of using second-order recursion is evident by noting that for the case of the radix-R FFT, the associated angles for each instance of CU_R are all multiples of some fixed angle and that multiplication by a twiddle factor equates to that of phase rotation. Thus, if the angle of the first non-trivial twiddle factor is given by φ_1 (noting that the value of the angle φ_0 of the initial trivial twiddle factor is always zero), then the remaining angles, $\{\varphi_m\}$, may be expressed as

$$\varphi_m = m \times \varphi_1 \tag{13}$$

for m = 2, …, R-1, which may be beneficially exploited when the trigonometric components of the twiddle factors are generated recursively via the appropriate *multi-angle formulae* [7], as given by

$$\sin(n\varphi_1) = 2 \times \sin((n-1)\varphi_1) \times \cos(\varphi_1) - \sin((n-2)\varphi_1) \tag{14}$$

$$\cos(n\varphi_1) = 2 \times \cos((n-1)\varphi_1) \times \cos(\varphi_1) - \cos((n-2)\varphi_1) \tag{15}$$

where ‘n’ is an arbitrary positive integer > 1.

Turning now to the question of complexity, the generation component of the solution – which is ideally suited to dealing with those big-data applications [13] where the on-chip storage capacity is limited – involves a multiplicative complexity of

$$C_{MULTS}^{(2,G)} = 8 \tag{16}$$

real multipliers (where each of the four multiplications required of the two-level LUT is followed by a simple hard-wired left-shift operation, of length one, at negligible cost), together with an associated additive complexity of

$$C_{ADDS}^{(2,G)} = 14 \tag{17}$$

real adders, this including the requirement for the address updating of the four complex-valued samples stored within the data memory and the four trigonometric component pairs stored with the LUTs, which are to be appropriately combined, and a trigonometric memory requirement of

$$C_{STORE}^{(2,G)} = 3 \times \sqrt{N/4} \tag{18}$$

words of fast RAM for the construction of the coarse-resolution and fine-resolution LUTs.

With regard to the application component of the solution, once the complex-valued data samples and the trigonometric component pairs of the three non-trivial twiddle factors have been generated, it simply remains for them to be appropriately combined. This involves a multiplicative complexity of

$$C_{MULTS}^{(2,A)} = 3 \times 3 = 9 \tag{19}$$

real multipliers, with the associated additive complexity of

$$C_{ADDS}^{(2,A)} = 3 \times 5 + 16 = 31 \tag{20}$$

real adders, this including 16 adders for carrying out the dragonfly's remaining two stages of pre/post (according to choice of decimation scheme) additions.

As a result, the total complexity of the dragonfly for this second solution, which includes the combined task of both generating and applying the twiddle factors, may be expressed as

$$C_{MULTS}^{(2)} = C_{MULTS}^{(2,G)} + C_{MULTS}^{(2,A)} = 17 \tag{21}$$

real multipliers, and

$$C_{ADDS}^{(2)} = C_{ADDS}^{(2,G)} + C_{ADDS}^{(2,A)} = 45 \tag{22}$$

real adders, and

$$C_{STORE}^{(2)} = C_{STORE}^{(2,G)} = 3 \times \sqrt{N/4} \tag{23}$$

words of fast RAM.

3.3 Discussion

The two conventional dragonfly solutions discussed in this section are based upon recently published results [6,7] which highlighted the relative merits of different schemes – using different combinations of single-level and multi-level LUTs, second-order recursion, adders and multipliers – for the efficient parallel generation, *but not the application*, of the non-trivial twiddle factors for the fixed-radix version of the FFT. These are just two possible solutions to the problem, however, as other combinations of LUTs, adders and multipliers could also be used at comparable cost to achieve similar results. Of the two solutions considered, Solution I sought to minimize the arithmetic complexity at the expense of an increased memory requirement, whilst Solution II sought to minimize the memory requirement at the expense of increased arithmetic complexity – noting that the recursion itself involved the use of just adders and multipliers. The attraction of the second solution, compared to the first, is evident when the transform length is sufficiently large and/or the on-chip storage capacity limited, this being due to the relative $O(L\sqrt{N})$ and $O(LN)$ complexities (including word length L due its relevance in describing complexity of CORDIC solution in Section 5) of their trigonometric memory requirements.

4. An Alternative Approach to Solving Problem of Phase Rotation

The previous two solutions have possessed complexities that are based upon the use of one or more LUTs, together with second-order recursion for one of the solutions, plus additional adders and multipliers, which taken together – depending upon the size of the transform – might prove prohibitively expensive in terms of the resulting silicon area. The third solution, as will be described in some detail in Section 5, will illustrate how the reliance upon the use of potentially costly LUTs may be effectively eliminated. Unlike the other two solutions, it will no longer involve having to perform two separate tasks, namely: 1) the generation of the trigonometric component pairs of the non-trivial twiddle factors, followed by 2) their multiplicative application to the complex-valued data samples, but will instead use: 3) multiple pipelined CORDIC arithmetic units which apply the *phase rotations* corresponding to the complex twiddle factor multiplications, piece-by-piece and in an iterative fashion – which, for a pipelined implementation, needs to be fully *unfolded* – whilst those pieces, which each correspond to a small-angle rotation, are actually being computed. The *generation and application stages are thus carried out simultaneously*, with the memory requirement being reduced to that required for the storage of a small number of angles, consistent with the word length, as required for execution of the small-angle rotations.

4.1 The Basic CORDIC Algorithm

As already stated, the basic operation of the fixed-radix FFT is essentially that of phase rotation arising from multiplication of the data by the associated twiddle factors, making the choice of the CORDIC algorithm an obvious candidate for consideration as the chosen arithmetic unit. The original version of the algorithm, as introduced by Volder [10], may be expressed as

$$x_{n+1} = x_n - d_n.y_n.2^{-n} \quad (24)$$

$$y_{n+1} = y_n + d_n.x_n.2^{-n} \quad (25)$$

$$\theta_{n+1} = \theta_n - d_n.\tan^{-1}\left(2^{-n}\right) \quad (26)$$

for n = 1,2, … , L where L specifies the accuracy to which the algorithm is to be carried out – noting that each additional iteration produces approximately one extra bit of accuracy in the rotated result – and where (x_0,y_0) represents the complex-valued sample of data to be rotated by a specified angle, denoted θ, and where the term θn determines whether the latest small rotation angle, ϕ_n, as given by

$$\phi_n = \tan^{-1}\left(2^{-n}\right), \quad (27)$$

is to be added to, or subtracted from, the current value of the cumulative rotation angle. The sequence of arctangent terms, which may be pre-computed and stored, are chosen for use by the CORDIC algorithm as the application of the corresponding tangents involves just simple bit-shift operations which may be hard-wired in silicon, without recourse to the use of a costly multiplier or barrel shifter, at negligible cost in terms of silicon area and/or power consumption.

Thus, when the CORDIC algorithm is operated in *rotation mode* (noting that it may be operated in various modes, other than that of rotation), each 'sense' term, d_n, may be expressed using the conditional statement signified by the 'sign' function

$$d_n = \text{sign}(\theta_n) \quad (28)$$

which yields a value of +1 for a positive argument and -1 for a negative argument, these terms enabling the small-angle rotations to be applied in the correct direction. The angle θ_0 is set equal to the required rotation angle, θ, which may be decomposed into the sum of smaller rotation angles

$$\theta = \sum_{n=0}^{\infty} d_n.\phi_n \quad (29)$$

so that rotation of a complex number (x_0,y_0) by an angle θ may be carried out in a straightforward recursive fashion as a sequence of elementary or small-angle rotations by angles $d_n.\phi_n$, such that

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} \cos\left(d_n.\tan^{-1}\left(2^{-n}\right)\right) & -\sin\left(d_n.\tan^{-1}\left(2^{-n}\right)\right) \\ \sin\left(d_n.\tan^{-1}\left(2^{-n}\right)\right) & \cos\left(d_n.\tan^{-1}\left(2^{-n}\right)\right) \end{bmatrix} . \begin{bmatrix} x_n \\ y_n \end{bmatrix} \quad (30)$$

which, by exploiting standard properties of the trigonometric functions, reduces to

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \cos\left(\tan^{-1}\left(2^{-n}\right)\right) \begin{bmatrix} 1 & -d_n.2^{-n} \\ d_n.2^{-n} & 1 \end{bmatrix} . \begin{bmatrix} x_n \\ y_n \end{bmatrix} \quad (31)$$

where

$$\cos\left(\tan^{-1}\left(2^{-n}\right)\right) = 1 \Big/ \sqrt{1+2^{-2n}} \quad (32)$$

Thus, if

$$K_n = \prod_{k=0}^{n} 1 \Big/ \sqrt{1+2^{-2k}} \quad (33)$$

(which, as 'n' is increased, tends to a value of 0.60725294 to eight decimal figures) then the rotated output, upon the completion of L 'multiplication-free' iterations – whereby, for each such iteration, the application of the *multiplicative term* of Eqtn. 32 is suppressed – requires the application of the *multi-term product* of Eqtn. 33 with 'n' being set to a value of L.

The attraction of suppressing the application of the individual multiplicative terms is that it enables the small-angle CORDIC rotation operation to be expressed in a *greatly simplified form as*

$$\begin{bmatrix} x_{n+1} \\ y_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & -d_n.2^{-n} \\ d_n.2^{-n} & 1 \end{bmatrix} \cdot \begin{bmatrix} x_n \\ y_n \end{bmatrix} \tag{34}$$

involving just additions and shifts, so that the tasks required of the n^{th} iteration of the CORDIC algorithm, when operating in rotation mode, may be summarized as

$$x_{n+1} = x_n - d_n.y_n.2^{-n} \tag{35}$$

$$y_{n+1} = y_n + d_n.x_n.2^{-n} \tag{36}$$

$$\theta_{n+1} = \theta_n - d_n.\phi_n \tag{37}$$

where (x_0,y_0) is the complex-valued sample of data to be rotated, θ_0 is set equal to the required rotation angle θ, and where the 'sense' term is given by

$$d_n = \text{sign}(\theta_n). \tag{38}$$

Suppressing the application of the multiplicative terms in this way must subsequently be corrected for, however, in order that the correct L-bit results should be obtained. To achieve this, it is required that after all L multiplication-free iterations have been completed, the multi-term product K_L – which may be pre-computed and stored in binary form once the value of L has been decided upon and performed as a number of shift-and-add operations – needs to be applied to both the real and the imaginary components of each of the rotated outputs, so that when all of the dragonfly inputs have been processed there will be consistency between the output levels produced by the application of both the trivial and the non-trivial twiddle factors to the respective data samples. An alternative and lower-complexity approach to achieving this consistency in the output levels is possible, however, as will be discussed in the next section.

4.2 Discussion

Much work has been carried out on the development of the CORDIC algorithm since the original paper of Volder [10] and, in particular, in extending the number of: 1) algorithmic variations of the CORDIC algorithm so as to cater, for example, for higher radices that are able to handle more than just {-1,+1} or {-1,0,+1} as defining the range of possible values for the 'sense' parameter used by the non-redundant and redundant versions of the algorithm [8,14,15], respectively, which enables the number of iterations (and thus the latency of the solution) to be commensurately reduced but at the expense of a non-constant scale factor; as well as 2) possible applications, such as those involving signal and image processing, communication systems, robotics or 3D graphics [16], that are able to exploit and benefit from the computational attraction, in terms of flexibility, versatility and simplicity, of the CORDIC approach. These advances, combined with the provision of highly-optimized CORDIC arithmetic units (typically as IP cores) by most major FPGA manufacturers, make the CORDIC algorithm as relevant today as when it was chosen for use with the first generation of pocket calculators, as emerged from the introduction of the semiconductor industry in the early 1960s.

5. Scheme for Eliminating Dragonfly's Trigonometric Memory Requirement

The previous section has shown how CORDIC arithmetic may be effectively applied to the problem of phase rotation, as is required when multiplying data samples by twiddle factors within CU_R, reducing each problem from that of a single rotation to a number of small-angle rotations, each of which may be carried out in hardware using simple adders and hard-wired bit-shifters. For this third solution, referred to hereafter as **Solution III**, it is now seen how a non-conventional CORDIC-based implementation of the radix-4 FFT dragonfly, CU_4, may be straightforwardly obtained at *zero cost in terms of trigonometric memory*. This is achieved with three pipelined CORDIC arithmetic units combined with additional adders (that is, apart from its own adders) for dealing with the dragonfly's remaining two stages of pre/post (according to choice of decimation scheme) additions.

5.1 Application of CORDIC Arithmetic Unit to Radix-4 FFT Dragonfly

The angles corresponding to the three non-trivial twiddle factors required by the radix-4 FFT dragonfly, CU_4, are given here by $\theta^{(S)}$ for the single-angle case of first non-trivial twiddle factor, by $\theta^{(D)}$ for the double-angle case of second non-trivial twiddle factor and by $\theta^{(T)}$ for the triple-angle case of third non-trivialtwiddle factor. The k^{th} cumulative estimate of each twiddle factor rotation angle may be expressed in terms of the small rotation angles of Eqtn. 27, ϕ_n , as

$$\theta_k^{(A)} = \sum_{n=0}^{k} d_n^{(A)} . \phi_n \tag{39}$$

where 'A' can stand for 'S', 'D' or 'T', with $d_n^{(A)}$ standing for the 'sense' or direction of each small-angle rotation and the corresponding sets of multiplication-free iterative equations given by

$$x_{n+1}^{(A)} = x_n^{(A)} - d_n^{(A)} . y_n^{(A)} . 2^{-n} \tag{40}$$

$$y_{n+1}^{(A)} = y_n^{(A)} + d_n^{(A)} . x_n^{(A)} . 2^{-n} \tag{41}$$

$$\theta_{n+1}^{(A)} = \theta_n^{(A)} - d_n^{(A)} . \phi_n \tag{42}$$

for the single-angle, double-angle and triple-angle cases, so that each of the three sets of multiplication-free equations include the updating of the cumulative estimate of the respective rotation angle and involves three additions/subtractions and two bit-shift operations – as illustrated in Figure 6 which depicts a single stage of a pipelined implementation of the unfolded CORDIC-based solution. This illustration shows that prior to the updating of the cumulative estimates of the twiddle factor rotation angles, the current estimates must be used to determine the values of the three 'sense' terms using the conditional statement signified by the 'sign' function

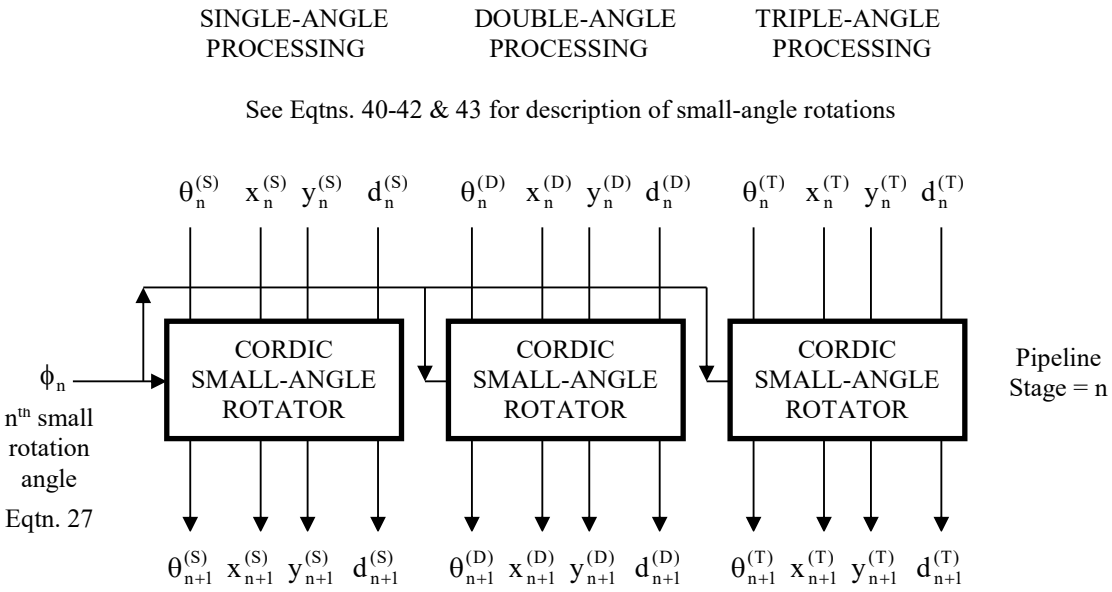

Figure 6: Stage of pipeline for CORDIC-based generation & application of twiddle factors for radix-4 FFT dragonfly

$$d_n^{(A)} = \text{sign}\left(\theta_n^{(A)}\right) \tag{43}$$

for the single-angle, double-angle and triple-angle cases.

5.2 Precision and Accuracy Considerations

Assuming the adoption of L multiplication-free iterations, each carrying out a small-angle rotation for each of the three non-trivial twiddle factor rotation angles, the scheme achieves approximately L-bit accuracy, with each additional iteration – which involves the maintaining and updating of nine parameters – as already stated, producing approximately one extra bit of accuracy in the rotated result. With each operation being assigned its own hardware operator, a pipelined implementation requires a total of nine adders and six hard-wired bit-shifters, for each stage, together with the storage of the set of L small rotation angles, $\{\phi_n\}$, as required for the updating of the cumulative estimates of the corresponding twiddle factor rotation angles. To deal with possible word growth incurred during the CORDIC operation, a number of guard bits are typically adopted, with L iterations requiring approximately $\lceil \log_2 L \rceil$ additional bits in order to prevent overflow and to maintain precision to the desired level.

Upon completion of all L multiplication-free iterations, the required L-bit accuracy may be achieved by having the multi-term product K_L applied to each of the three complex-valued outputs (involving six real-valued components), this involving the use of six constant multipliers. Alternatively, rather than applying the multi-term product to the rotated outputs obtained from the application of the non-trivial twiddle factors, the complex-valued output obtained from using the trivial twiddle factor (involving two real-valued components) could instead be modified by means of the inverse version of the multi-term product of Eqtn. 33, namely:

$$G_n = 1/K_n \tag{44}$$

(which, as 'n' is increased, tends to a value of 1.64676026 to eight decimal places), with 'n' set to a value of L, which would involve the use of just two constant multipliers, rather than six. In this way, each of the four complex-valued outputs on completion of the rotation operation would be in error by a multiplicative factor equal to that of the multi-term product, G_L, the effects of which may subsequently be corrected for, together with any naturally occurring word growth, after each of the $\log_4 N$ temporal stages of FFT dragonflies. This correction could be achieved by means of a suitably defined scaling strategy, such as that of the *block floating point scheme*, for example, as would typically be required by any fixed-point implementation of the fixed-radix FFT algorithm (regardless of the choice of arithmetic unit) for dealing with word growth and thus for maintaining accuracy [17].

5.3 Complexity of CORDIC-Based Solution

With this solution, assuming that questions relating to precision and accuracy have been suitably addressed, the total complexity involving both the arithmetic complexity and the trigonometric memory requirement, may be expressed as requiring

$$C^{(3)}_{ADDS} = 9\times L + 23 \tag{45}$$

real adders, this including the requirement for the address updating of the four complex-valued samples stored within the data memory as well as for the updating of the three rotation angles and the 16 adders required for carrying out the dragonfly's remaining two stages of pre/post (according to choice of decimation scheme) additions, together with

$$C^{(3)}_{SHIFTS} = 6\times L \tag{46}$$

hard-wired bit-shifters, which incurs negligible cost, plus

$$C^{(3)}_{MULTS} = 2 \tag{47}$$

constant real multipliers, and

$$C^{(3)}_{STORE} = L \tag{48}$$

words of fast RAM for storage of the set of L small rotation angles, $\{\phi_n\}$, which need to be distributed to each of the three pipelined CORDIC arithmetic units – the need for trigonometric memory is effectively eliminated. Control logic is also required for each iteration, this including six conditional statements with three for determining the 'sense' of each of the three $d_n^{(A)}$ terms required for the corresponding sets of equations and three for subsequently applying them – although when compared to the other complexity

components this contribution, as with that of the hard-wired bit-shifters, is taken to be negligible.

5.4 Discussion

With the CORDIC-based solution described here there is no longer a requirement to construct and maintain potentially large LUTs for the storage of the twiddle factors and that for the operation of phase rotation the CORDIC approach may be considered as being close to optimal in achieving high precision and fast convergence for minimum cost in terms of hardware resources. Such an approach possesses the attraction of great versatility, as the arithmetic unit can be designed to range from being: 1) *very small*, in terms of silicon area, through the adoption of a slow, iterative, purely sequential solution, to 2) *very fast*, in terms of throughput, via the adoption of a fully pipelined version of the unfolded solution. With regard to its efficient implementation, the use of conventional barrel shifters for carrying out the bit-shifting would involve both high complexity and high latency, whereas that of hard-wired bit-shifters, as has been proposed here, results in negligible cost, lower latency and higher throughput. As a result, the complexity of the radix-4 FFT dragonfly, CU_4, is dominated by the adder requirement of Eqtn. 45, with the constant multipliers, in comparison, making a negligible contribution.

Note that the version of the CORDIC algorithm discussed here is referred to in the literature as being *non-redundant* due to the fact that by setting the value of each d_n to either +1 or –1, each small rotation angle is always applied according to the associated sense, even if it is not absolutely necessary to do so – that is, if it produces a less accurate estimate than the previous iteration. An alternative version of the algorithm, referred to as being *redundant*, allows the value of d_n to be set to 0, as well as +1 or -1, so that specific small rotation angles may, if necessary, be ignored. As a result of this distinction, the *latency* of a pipelined implementation of the redundant version of the CORDIC algorithm may vary according to the value of the original rotation angle whereas the non-redundant version possesses the attractive property that the latency is fixed, being independent of the original rotation angle. Clearly, when multiple pipelined CORDIC arithmetic units are required to operate in parallel, it is essential that they should possess the same latency in order to ensure synchronization, otherwise angle dependent delays might be needed to ensure that their outputs are simultaneously available for subsequent processing.

6. Complexity Comparison of All Three Solutions

Adopting the silicon-based complexities of the computational building blocks, as introduced in Section 2, the overall sizing of Solution I for the case of a single highly-parallel FFT dragonfly, CU_4, based upon the use of single-level LUTs combined with adders and multipliers, may be expressed as

$$C^{(1)} = 9 \times L^2 + \left(3N/4 + 19\right) \times L \tag{49}$$

logic slices, whilst that of Solution II, based upon the use of a two-level LUT and second-order recursion combined with additional adders and multipliers, may be expressed as

$$C^{(2)} = 17 \times L^2 + \left(3\sqrt{N/4} + 45/2\right) \times L \tag{50}$$

logic slices, and that of Solution III, based upon the use of three pipelined CORDIC arithmetic units combined with additional adders, may be expressed as

$$C^{(3)} = \left(11/2 \times L + 27/2\right) \times L \tag{51}$$

logic slices – this due mainly to the contribution of the 9×L array of adders, as given by Eqtn. 45.

These results, which may be summarized in a succinct fashion by simply stating that Solution I possesses $O(LN)$ complexity, Solution II $O(L\sqrt{N})$ complexity and Solution III $O(L^2)$ complexity, where N >> L, also generalize in an obvious fashion to the case of CU_R, for the radix R FFT, where R is an arbitrary positive integer.

The complete set of complexity results are as listed in Tables 1 and 2, for various combinations of transform length and word length, these being based upon a highly parallel implementation of a single PE performing the radix-4 FFT dragonfly, CU_4, using programmable logic slices – although it is anticipated that the relative advantages/disadvantages of each solution would apply equally when implemented using the highly optimized embedded functions made available by the device manufacturer. Of the two conventional solutions discussed, Solution I sought to minimize the arithmetic complexity at the expense of an increased memory requirement, whilst Solution II sought to

minimize the memory requirement at the expense of increased arithmetic complexity. The complexity of Solution III, unlike that of the other two solutions, is dependent upon the word length but independent of the transform length so that the solution, as well as looking attractive for the short-to-long transforms, looks increasingly attractive as the transform length is increased. Thus, for those big-data applications where resources, such as fast RAM for the trigonometric memory, may be limited, the trivial memory requirement of the CORDIC-based approach makes it look particularly attractive in terms of reduced silicon area. With Solutions I and II, when the memory requirement is sufficiently large, it is possible that slower *off chip* memory might well be needed to supplement the on chip RAM which could, as a result, lead to potential timing issues.

Resource Type (L-bit Operators)	Resources for Dragonfly of Length-N Radix-4 FFT		
	Solution I	Solution II	Solution III
multipliers	9	17	2 (constant)
adders	38	45	9×L+23
trigonometric memory (words)	$3\times N/4$	$3\times\sqrt{N/4}$	L
Hard-wired bit-shifters	-	-	6×L (negligible cost)

Table 1: Resources for length-N radix-4 FFT dragonfly with L-bit fixed-point processing

FFT Length	Word Length (bits)	Complexity of Dragonfly for Length-N Radix-4 FFT (operators + trigonometric memory ~ slices)		
		Solution I	Solution II	Solution III
N = 1024 (Long)	L = 16	0.149×10^5	0.548×10^4	0.160×10^4
	L = 24	0.241×10^5	1.148×10^4	0.346×10^4
	L = 32	0.344×10^5	1.965×10^4	0.602×10^4
$N = 1024^2$ (Very Long)	L = 16	0.126×10^8	0.293×10^5	0.160×10^4
	L = 24	0.189×10^8	0.472×10^5	0.346×10^4
	L = 32	0.252×10^8	0.673×10^5	0.602×10^4
$N = 1024^3$ (Ultra Long)	L = 16	0.129×10^{11}	0.792×10^6	0.160×10^4
	L = 24	0.194×10^{11}	1.190×10^6	0.346×10^4
	L = 32	0.258×10^{11}	1.591×10^6	0.602×10^4

Table 2: Complexity for length-N radix-4 FFT dragonfly with L-bit fixed-point processing (control logic & hard-wired bit-shifters not taken into account)

Note, finally, that the complexity of the associated radix-4 FFT would clearly be dependent upon the chosen computing architecture, whether to be based, for example, upon the adoption of 1) a *scalable memory-based architecture*, involving the repeated execution of a single highly parallel PE exploiting *fine-grain parallelism* via the use of spatial-domain and/or temporal domain techniques; or 2) a *pipelined architecture*, exploiting *coarse-grain parallelism* via the simultaneous execution of multiple PEs – for the radix-R case, this would entail the use of $\log_R N$ such PEs, one per stage, with each stage catering for N/R dragonflies. Double buffering of some kind would probably be needed in each case in order to prevent the occurrence of unnecessary delays in the processing, With these and other hybrid approaches, each would have its own merits according to the values of the various parameters involved, such as that of transform length and word length, as well as to the constraints of the particular application, such as update period, clock frequency, power budget, memory capacity (in the form of fast RAM), etc. With regard to the actual PE, it should be noted that with an arbitrary radix, R, a total of R-1 non trivial twiddle factors would need to be generated and applied to the corresponding data set. This suggests, not surprisingly, that the larger the radix, the greater the amount of parallelism that might be exploited by the individual PE given a suitably chosen FFT computing architecture combined with a suitable PE design.

7. Summary and Conclusions

This paper has analyzed the relative complexities, as expressed in terms of arithmetic and trigonometric memory requirements, of three different solutions to the parallel computation of those arithmetic tasks needing to be performed by the radix-4 FFT dragonfly, CU_4, for different combinations of transform length and word length. The two conventional solutions were based upon the combined use of LUTs, adders and multipliers: Solution I, employing three single-level LUTs combined with adders and multipliers, sought to minimize the arithmetic complexity at the expense of an increased memory requirement, whilst Solution II, employing a two-level LUT and second-order recursion combined with additional adders and multipliers, sought to minimize the memory requirement at the expense of increased arithmetic complexity. Solution III took a non-conventional approach, being based upon the use of three pipelined CORDIC arithmetic units combined with additional adders and had a complexity, unlike that of the other two solutions, that was dependent upon the word length but independent of the transform length, effectively eliminating the trigonometric memory requirement at the expense of an increased number of adders. These properties follow directly from the derived complexity results which stated that **Solution I** possesses $O(LN)$ complexity, **Solution II** $O\left(L\sqrt{N}\right)$ complexity and **Solution III** $O\left(L^2\right)$ complexity, where N >> L, with the results generalizing in an obvious fashion to the case of CU_R, for the radix R FFT, where R is an arbitrary positive integer. Thus, the results of the study have shown, in particular, that the CORDIC-based approach of Solution III looks increasingly attractive as the transform length is increased, especially for those big-data applications where resources, such as fast RAM, may be limited. The solution relies upon the use of a potentially large number of adders, however, so that the key to an efficient implementation of Solution III being realized – assuming the CORDIC unit is designed from 'scratch' as opposed to being a highly-optimized IP core, as made available by the device manufacturer – is that a hardware efficient adder design should be identified that's able to effectively exploit the properties of silicon based technology as typified by the FPGA [18,19].

Conflicts of Interest: The author, having been retired for some years, has no access to funding of any kind and states that there are no conflicts of interest associated with the production of this paper.

References

1. E.O. Brigham. (1974). "*The Fast Fourier Transform*". Prentice-Hall.
2. E. Chu & A. George. (2000). "*Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*". CRC Press.
3. Ahmed, N., & Rao, K. R. (2012). "*Orthogonal transforms for digital signal processing*". Springer Science & Business Media.
4. G. Birkhoff & S. MacLane. (1977). "*A Survey of Modern Algebra*". Macmillan.
5. K. Jones. (2026). "*A Taxonomy of Solutions for Fixed-Radix FFT Algorithms*". Engineering (OPAST Open Access), Vol. 4, No. 3.
6. K. Jones. (2024). "*Schemes for Resource-Efficient Generation of Twiddle Factors for Fixed-Radix FFT Algorithms*". Engineering (OPAST Open Access), Vol. 2, No. 3.
7. K. Jones. (2025). "*A Comparison of Two Schemes, Based upon Multi-Level LUTs and Second Order Recursion, for Parallel Computation of FFT Twiddle Factors*". Trans. on Applied Science, Engineering and Technology (Open Access), Vol. 1, No. 1.
8. Andraka, R. (1998, March). "A survey of CORDIC algorithms for FPGA based computers". In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays* (pp. 191-200).
9. J.M. Muller. (1997). "*Elementary Functions: Algorithms and Implementation*". Birkhauser.
10. J.E. Volder. (1959). "The CORDIC Trigonometric Computing Technique". IRE Trans. on Electronic Computing, Vol. EC-8, No. 3, pp. 330-334.
11. C. Maxfield. (2004). "The Design Warrior's Guide to FPGAs". Newnes (Elsevier).
12. S.G. Akl. (1989). "The Design and Analysis of Parallel Algorithms". Prentice-Hall.
13. K. Jones. (2024). "Design of Scalable Architecture for Real-Time Parallel Computation of Long to Ultra Long Real-Data DFTs". Engineering (OPAST Open Access), Vol. 2, No. 4.

14. Bhattacharyya, K., Hazra, A., Hatai, I., & Banerjee, S. (2009). "Architectural design of a Radix-4 CORDIC-based Radix-4 IFFT algorithm and its FPGA implementation". *International Journal of Signal and Imaging Systems Engineering, 2*(4), 201-215.
15. R. Lakshme & A.S. Dhar. (2010). “CORDIC Architectures: A Survey”. VLSI Design, ID 794891.
16. Meher, P. K., Valls, J., Juang, T. B., Sridharan, K., & Maharatna, K. (2009). "50 years of CORDIC: Algorithms, architectures, and applications". *IEEE Transactions on Circuits and Systems I: Regular Papers, 56*(9), 1893-1907.
17. Altera Corporation. (2005). “FFT/IFFT Block Floating Point Scaling”. Application Note PDF, San Jose, CA, USA.
18. Kogut, I., Hryha, V., Dzundza, B., Hryha, L., & Hatala, I. (2024). "Research and Design of Multibit Binary Adders on FPGA". *Advances in Cyber-Physical Cystems, 9*(2), 108-114.
19. Fayaz Begum, S., Kavya Sree, M., Amzadhali, S., Venkata Sai Sushma, J., & Sai Kumar, S. (2023, March). "Analysis of the Efficiency of Parallel Prefix Adders". In *International Conference on Communications and Cyber Physical Engineering 2018* (pp. 105-118). Singapore: Springer Nature Singapore.