Any-Radix Efficient Fully-Parallel Implementation of the Fast Fourier Transform on FPGAs

Ignacio Amat Hernández
Dpto. Ingeniería Electrónica
Univ. Politécnica de Madrid
Madrid, Spain
ignacio.amat@alumnos.upm.es
ORCID: 0009-0001-4624-5623

Juan A. López
Dpto. Ingeniería Electrónica
Univ. Politécnica de Madrid
Madrid, Spain
juanantonio.lopez@upm.es
ORCID: 0000-0002-5808-5014

***Abstract*—This paper presents the results of a thorough investigation of arbitrary and mixed-radix architectures of the Fast Fourier Transform (FFT) and their efficient implementation on high performance FPGA devices. We have developed a novel methodology that recursively generates the hardware description of the logic architecture of the FFT using parameterized complex matrix multipliers. Our approach allows to easily and efficiently generate fully-parallel implementations of an arbitrary number of points. This is used for example in state-of-the-art systems with non-power-of-two number of points, such as some 6G mobile communications systems. In addition, the proposed architectures accept large amounts of pipeline to meet rigorous timing requirements. The largest implementation provided surpasses the 100 GS/s throughput threshold.**

***Index Terms*—Digital Signal Processing, Fourier Transform, FFT, FPGA, Hardware implementation**

I. Introduction and state of the art

Fourier Transforms (FT) are considered a staple of digital signal processing due to their wide range of applications. In particular, the FFT and its inverse (IFFT) are widely used to analyse and synthesize signals in many signal processing systems. Their fields o f a pplication r ange f rom engineering, physics and chemistry to computer science, medicine, biology and many others.

Some examples of more detailed real-world applications include equalization, error detection and noise reduction in digital communications [1], image enhancement, noise reduction and compression in image processing [2], computer tomography, magnetic resonance imaging and positron emission tomography in medical exploration and non-invasive visualization [3], stock options pricing and identification of hidden trends and patterns in financial analysis [4], astronomy and seismic analysis in scientific r esearch [5, 6], a nd many others.

Software implementations of the FT for Digital Signal Processing (DSP) on general-purpose microprocessors offer the highest flexibility with the least amount of implementation effort at the cost of throughput, resource and energy efficiency. Notable efforts include the *Fastest Fourier Transform in the West* (FFTW) software library by Frigo and Johnson [7], an open source parameterized FFT generator for a wide array of parameters and architectures. It consists of different algorithms for computing the Discrete Fourier transform (DFT) in one or more dimensions. It is one of the most popular and widely used libraries for computing the DFT in CPUs. FFTW is highly optimized for modern processors, it is also highly scalable, allowing to be used on a wide range of systems, from small embedded systems to large distributed many-core multi-node high performance computing platforms. FFTW is also highly portable, and can be used on a variety of platforms, including Linux, Mac OS X, and Windows.

The NVIDIA cuFFT [8] software toolkit is a GPU-accelerated library for computing FFTs. It is designed to provide high performance and scalability for a wide range of architectures. The library is optimized for use with multiple NVIDIA GPUs, and provides several sets of features, including support for single and double precision arithmetic, in-place and out-of-place transforms, and support for multidimensional transforms. It also provides a range of performance tuning options, including batching, tiling, and data layout optimisation. Similarly to FFTW, its library is designed to be easy to use, with a simple API and comprehensive documentation. It is also highly portable, with support for a wide range of operating systems and programming languages. The library is available as part of the NVIDIA CUDA Toolkit [9]. Though it is not open-source, like FFTW, it is free for academic and commercial use.

Several hardware topologies have been proposed to efficiently realise the Fourier transform in FPGAs and as ASICs. Most hardware architectures take advantage of two design techniques, spatial and temporal parallelism (i.e., time multiplexing and pipelining), to maximize its performance. Designs that rely heavily on time multiplexing techniques yield systems that work in series. They process one sample every clock cycle and require as many cycles as samples are needed to complete the computation. On the other hand, designs that apply spatial parallelism typically obtain parallel implementations. They process multiple samples per clock

This work was partially supported by MCIN/AEI/10.13039/501100011033 and "ERDF A way of making Europe" under Project PID2021-126991NA-I00.

cycle, and when fully paralelised all the samples are processed in a single clock cycle. Between these two extremes we find hybrid architectures. These architectures try to balance the amount of resources and the throughput by keeping some parts of the circuit in parallel and others in series. This trade-off allows for tailored implementations to meet the required specifications.

Serial hardware implementations boast a small footprint in terms of resources, at the expense of throughput. Pipelined implementations are built by concatenating butterfly processing elements alongside memory registers. Each unit is reused for the computation of each sample [10, 11]. Parallel implementations offer the highest possible throughput, but they also require more resources.

The main features presented in this work are as follows:

- A novel recursive methodology to develop arbitrary-size any-radix fully-parallel FFT architectures, its validation and implementation in high performance FPGAs.
- The incorporation of pipeline techniques to obtain the highest clock rates achievable on Virtex Ultrascale+ (891 MHz for speed grade -3).
- A comparison in terms of resource utilization among FFTs of different sizes, according to the number of points and the factorisation of the FFTs.

II. Theoretical Background

The DFT of N points can be compactly expressed in matrix form as the product of an N-length input vector, x, and an $N \times N$ transform matrix T_N:

$$DFT_N(x) = T_N \cdot x \tag{1}$$

where each entry $(i, j) = (n, k)$ of the transform matrix T_N corresponds to the complex-valued Fourier coefficient, computed as:

$$T_{N\,ij} = e^{-j\frac{2\pi}{N}nk} = W_N^{nk} \tag{2}$$

$$T_N = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N & W_N^2 & \cdots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & \cdots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \cdots & W_N^{(N-1)(N-1)} \end{pmatrix} \tag{3}$$

In linear algebra [12], equation (3) is an example of a Vandermonde matrix, named after Alexandre-Théophile Vandermonde, who introduced it in 1771 while studying the interpolation of polynomials [13]. A Vandermonde matrix V is a matrix that has the terms of a geometric progression in each row. The elements $V_{i,j} = x_i^{j-1}$, where $x_1, x_2, ..., x_n$ are the elements of a given vector $\mathbf{x}$. For example, when $\mathbf{x} = [1, 2, 3]$, then the corresponding Vandermonde matrix is:

$$V_{\mathbf{x}} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix} \tag{4}$$

In this case, equation (3) is the Vandermonde matrix of the roots of unity. A root of unity is a complex number z such that $z^n = 1$ for some positive integer n. For example, when $n = 2$, the roots of unity are 1 and -1. When $n = 3$, the roots of unity are $z = 1$, $\frac{-1+\sqrt{3}j}{2}$, and $\frac{-1-\sqrt{3}j}{2}$ and for $n = 4$, the solutions to $z^4 = 1$ are $1, j, -1, -j$. These roots of unity have important properties and applications in various fields of mathematics, including algebraic geometry and number theory [14]. They also have applications in physics, such as in the study of quantum mechanics [15]. In addition to its applications in interpolation [16], the Vandermonde matrix also has applications in linear algebra and numerical analysis, such as the computation of the matrix inverse and the solution of linear systems [13], the matrix inverse of (3) corresponds to the inverse DFT.

Using the $n = 2$ roots of unity, the two-point discrete Fourier transform matrix T_2 is:

$$T_2 = \begin{pmatrix} 1+0j & 1+0j \\ 1+0j & -1+0j \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{5}$$

As is evident from this matrix representations of the different Fourier transforms, the computation of an N-length DFT requires a vector-matrix product of size $(N \times N) \cdot (1 \times N)$. This has a computational complexity of the order of $\mathcal{O}(N^2)$, far from ideal. A single point of an N-point DFT requires N complex multiplications and $N - 1$ complex additions. An N point DFT requires N^2 complex multiplications and $N \cdot (N-1)$ additions.

In addition, by inspection we see that in some cases the Fourier transforms are composed primarily of trivial multiplications, mainly by ± 1 and $\pm j$. This is of great importance, since implementation of such multiplications require negligible hardware.

Efficient Fourier implementations can be obtained by carefully factorising the transform matrix to our advantage. Thanks to the symmetry of this system, we can minimize the number of operations that are repeated. These factorisations are shown in works by Pease [17], Sloate [18], Cortés et al. [19], and produce the following recursive matrix form Fourier transform which expresses the Cooley and Tukey [20] algorithm.

$$T_N = P_N^{(r)} \cdot [I_r \otimes T_{N/r}] \cdot D_N^{(r)} \cdot [T_r \otimes I_{N/r}] \tag{6}$$

The symbol $\otimes$ denotes the Kroenecker product, the generalization of a scalar-matrix multiplication. $P_N^{(r)}$ is the permutation matrix. D_N^r corresponds to the diagonal matrix containing the Fourier coefficients. $I_{N/r}$ is the N/r identity matrix, where N is the number of points, and r is the radix.

This generalises the Fourier transform for any N number of points which are calculated recursively as any arbitrary smaller transforms of size N/r. The total number of operations is reduced leading to complexities of the order of $\mathcal{O}(N \cdot \log_2(N))$ in some cases.

The complexity concisely expressed in (6) is the key that enables us to very naturally and very simply generate parallel

$$T_4 = \begin{pmatrix} 1+0j & 1+0j & 1+0j & 1+0j \\ 1+0j & 0-1j & -1+0j & 0+1j \\ 1+0j & -1+0j & 1+0j & -1+0j \\ 1+0j & 0+1j & -1+0j & 0-1j \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} \tag{7}$$

hardware implementations of factorised Fourier transforms. Thanks to this expression we can define the digital logic architecture for an arbitrary number of points with an arbitrary number of factorisation stages without adding any computational overhead to the Cooley and Tukey algorithm [20].

III. Proposed Approach

The 4-point Fourier transform of (7) can be factorised according to (6) to produce (8) using the 2-point Fourier transform given by (5). This 4-point Fourier decomposition illustrates the expected particularly efficient behaviour, as all entries of the matrices are ± 1 or $\pm j$ they can be implemented with very little hardware. Higher order Fourier transforms are then built using deeper levels of recursion. In expansion (8), the first matrix corresponds to the permutation matrix, which can be implemented in hardware by carefully ordering signal connections without needing a matrix multiplication. The second matrix corresponds to the $[I_r \otimes T_{N/r}]$ term of (6), for which $T_{N/r}$ is the transform matrix that has to be calculated recursively. The next matrix is $D_N^{(r)}$, which contains the Fourier coefficients given by (2). The last matrix is $[T_r \otimes I_{N/r}]$, which is dependent on the number of points, N, and the radix, r, of each stage. The entries of the last two matrices are precomputed and added to our designs as constants. This allows the implementation tool to simplify trivial multiplications and eliminate multiplications by zero entirely.

Fig. 1 shows the four inferred RTL blocks generated for the multiplication by the decomposition of matrix T_4. It features two matrix multiplications which are then divided into two T_2 matrices. The output of these signals is then rearranged as to satisfy the permutation requirements. This can be described in VHDL without requiring an additional matrix multiplication. The call to the lower order FFT is done recursively, so the same module instantiates itself with different values for the generic parameters.

The proposed systems operate in parallel. An arbitrary N-point FFT takes N input samples within the same clock cycle and generates N outputs simultaneously after the pipeline number of cycles of latency. This aggressive approach represents the first step towards achieving the maximum processing throughput. Higher clock rates are achieved adding pipeline registers to break up the critical combinational path.

To appropriately define the architecture of an arbitrarily-sized Fourier transform of any radix, a matrix-based approach has been derived. The system is recursively instantiated in VHDL using complex matrix multipliers. The constants for each of the matrices, containing the Fourier coefficients are generated using a Python script. This is one of the key developments of our approach, since it maximizes the flexibility of the process.

The output of the multiplications are truncated to maintain the integer and fractional word-lengths of our datapath, i.e., an $n \times n$ multiplication yields an n-bit wide result. Both, signals and coefficients, are quantised to 18 bits for each of the real and complex parts, yielding 36-bit wide samples. The real and complex parts are formatted to signed fixed-point representation with 12 bits of precision for the fractional part, and truncation is performed accordingly. Word-lengths of 18 bits per part are chosen by default, since this is the size of the smallest port of the DSP48E2 block, LUT implementations remain at a standard 16 bits per part. However, the architecture has been parametrized to accept any arbitrary width for any of the signals, including a different width for the coefficients. Similarly, any arbitrary fixed-point representations can be set as a parameter.

The complex multiplier strategy uses 4 real multiplications and 2 additions. For two complex numbers, $a = x + y \cdot j$ and $b = u + v \cdot j$, conventional multiplication is defined as $a \cdot b = (x + y \cdot j) \cdot (u + v \cdot j) = (xu - yv) + (xv + yu) \cdot j$.

To achieve the maximum clock rate and maximise the throughput, we indicate the implementation tool to implement trivial multiplications as distributed logic using LUTs and non-trivial real multiplications on DSP slices. This compromise solution shares the utilization load between the logic fabric and available DSP slices. Though it might seem inefficient to use DSP multipliers for signal-constant multiplication, the total number of DSPs occupied represent a small percentage of the total available. Additionally, the proposed architecture can easily be adapted for LUT only implementation if an area-efficient optimisation is needed, though this might come with optimisation penalty or an increase of pipeline registers.

All transforms have been tested on a Virtex Ultrascale+ speed grade -3 FPGA (XPCVU35P-FSVH2104-3-E) using the synthesis strategy *Flow Alternate Routability* and the implementation strategy *Performance Explore Post Route Phys. Opt.* The target frequency was set to 891 MHz (clock period 1.123 ns.), which is the maximum clock rate supported by the FPGA. The speed grade -3 device has the fastest clock of the FPGAs currently supported by Xilinx. This FPGA features 872k Look Up Tables, 1743k D-type Flip Flop registers and 5952 DSP48E2 Arithmetic Logic Units. DSP48E2 slices feature an 18×27 bit signed multiplier and a pre- and post-adder that performs addition and subtraction, as well as other logic operations. When this slice is fully pipelined, it achieves the aforementioned 891 MHz clock rate.

The constant complex dot product generator features a

$$T_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -j \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \tag{8}$$

$$T_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} T_2 & & 0 & 0 \\ & & 0 & 0 \\ 0 & 0 & & \\ 0 & 0 & & T_2 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -j \end{pmatrix} \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \tag{9}$$

$$T_N = T_{perm} \times T_{recur} \times T_{diag} \times T_{kron} \tag{10}$$

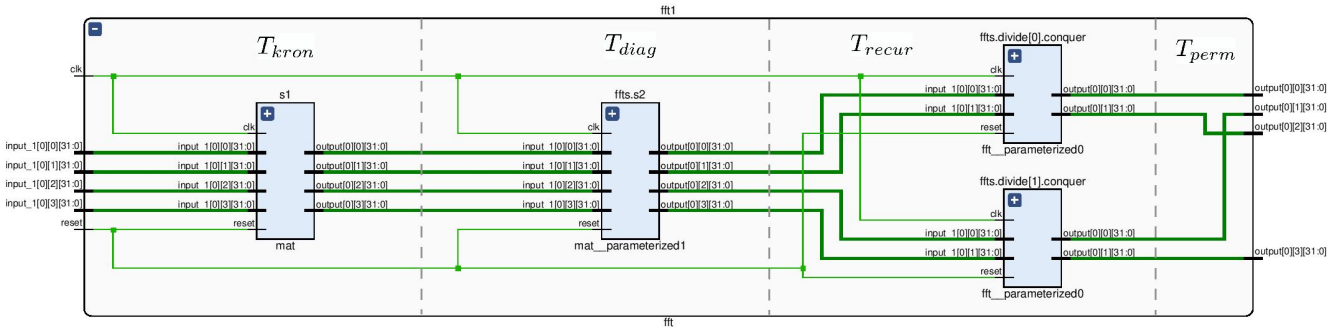

Fig. 1. Inferred factorised matrices generated to perform a 4-point radix-2 FFT. The first block corresponds to the multiplication between the input vector and matrix T_{kron}. The resulting vector becomes the input of the multiplication by T_{diag}. Next, T_{recur} is recursively instantiated the required number of times. Finally, T_{perm} is automatically inferred by the connections to the output ports.

generic adder tree following the complex multiplications. This reduces the critical path in comparison to linearly chaining adders. During implementation, the multiplications by zero are substituted by an `IF` statement, which results in a more stream-lined hardware. Throughout the hardware descriptions, groups of registers have been added to breakup the critical signal paths. This can be seen at the outputs of the scalar product and withing the constant complex multipliers.

The most efficient complex multiplier observed with the proposed setup is the four multiplier approach. The result is truncated at the output stage, but the location of the binary point is maintained at the same position so as not to alter the magnitude of the output number.

LUT-based multiplications are optimised by using generic Canonical Signed Digit multipliers. The non-adjacent form of the constant is precomputed with the Prodinger [21] method. This approach seeks to reduce the complexity of a multiplication by using both positive and negative partial products.

DSP systems have 3 cycles of latency per complex matrix multiplication, i.e., 6 latency clock cycles per stage of the decomposition. LUT-based systems require 5 clock cycles per matrix multiply, which results in a total of 10 clock cycles of added latency per stage.

The values of the constant matrices are calculated with a simple Python script using `numpy` [22] functions. The values are written in a file and read from the VHDL source code.

The developed systems have been tailored for throughput, though in future work area optimisation will also be investigated. We match our surpass the throughput of state of the art parallel FFT implementations at the cost of an increase in resource usage. On top of this, the proposed architecture also covers FFT sizes and radices not commonly found in scientific literature. They are simulated with a test signal and the results are compared to the expected values calculated using the numpy library FFT function. The noise is calculated as the difference between both results, and used to calculate the Signal-to-Quantization-Noise ratio. An example of the results obtained for a 64-point FFT is given in Fig. 2.

IV. Results and Discussion

A. DSP-based FFT implementations

Table I summarises the characteristics of the developed systems. The Signal-to-Noise ratio is approximated by generating an input signal, simulating the circuit response and calculating the noise as the difference between the obtained

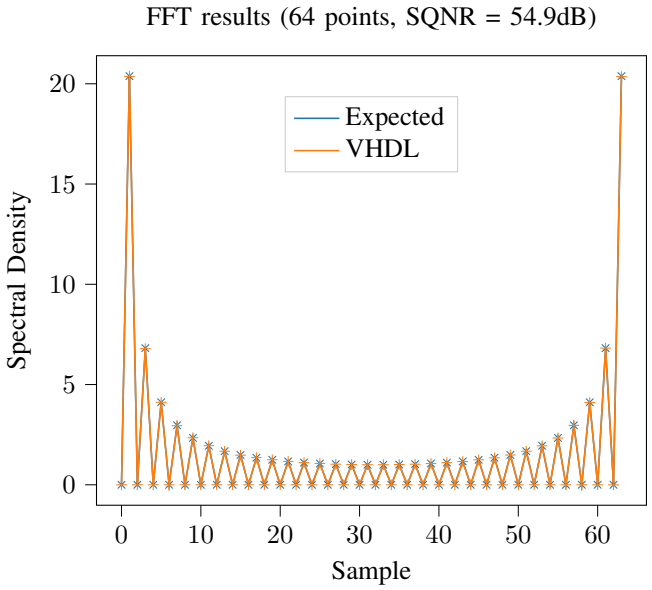

Fig. 2. Comparison between the results provided by our hardware architecture and the floating-point simulation of a 64-point FFT.

TABLE I
DSP-BASED FFT IMPLEMENTATIONS ON VIRTEX ULTRASCALE+

Points	Factorisation	F_{max} (MHz)	LUT	FF	DSP	SQNR (dB)	GS/s
12	3*4	891	3.9k	12.1k	40	62	10.69
15	5*3	724	8.0k	19.4k	264	63	10.86
16	4*4	891	5.3k	14.8k	30	75	14.25
18	3*3*2	718	8.1k	24.7k	144	66	12.92
24	2*2*2*3	891	10.8k	38.3k	127	70	21.38
27	3*3*3	744	14.9k	42.1k	328	66	20.08
30	5*6	648	25.1k	49.5k	694	66	19.44
32	4*4*2	891	13.6k	42.2k	106	71	28.51
36	3*3*4	715	19.4k	55.5k	288	67	25.74
48	2*2*2*2*3	829	29.8k	88.5k	333	69	39.70
48	2*4*3	843	25.9k	68.5k	266	71	40.46
54	3*3*3*2	678	35.9k	107.5k	708	64	36.61
64	2*2*2*2*2*2	891	34.4k	119.6k	377	68	57.16
64	2*2*2*2*4	817	33.9k	109.0k	377	68	53.16
64	4*4*4	862	35.1k	97.0k	294	69	55.16
81	3*3*3*3	651	60.8k	161.6k	1408	56	52.73
96	2*2*4*2*3	824	63.2k	208.0k	841	59	79.10
128	4*4*4*2	788	82.8k	245.4k	790	62	100.23
128	2*2*2*2*2*2*2	796	81.5k	283.4k	1001	60	101.88

output and the expected result. This has been non-exhaustively done using only one signal in each case, so it can only be taken as a coarse estimate. Future work is needed to more precisely measure the signal characteristics.

The measured trends are in concordance with the expected results. The power-of-two Fourier transforms require fewer DSP multipliers, since they also require fewer non-trivial multiplications. The radix-3 decompositions require more DSP blocks and resources, and they also achieve poorer throughput. Overall, regardless of the radix, we see an increase in the number of resources as the number of points increase. Despite the increase in complexity, our architecture and the pipelined realisation manage to maintain very high clock rates. Our 128-point parallel FFT achieves a 100 GS/s throughput both in

TABLE II
LUT-BASED FFT IMPLEMENTATIONS ON VIRTEX ULTRASCALE+

Points	Factorisation	F_{max} (MHz)	LUT	FF	DSP	SQNR (dB)	GS/s
16	2*2*2*2	891	7.9k	28.2k	0	75	14.25
16	4*4	891	8.3k	22.8k	0	75	14.25
24	2*2*2*3	891	20.5k	59.4k	0	72	21.38
27	3*3*3	738	38.5k	92.7k	0	60	19.92
32	2*2*2*2*2	891	22.6k	67.2k	0	65	28.51
32	4*4*2	891	22.9k	62.0k	0	65	28.51
48	4*4*3	708	46.8k	126.7k	0	70	33.98
64	2*2*2*2*2*2	794	59.2k	166.3k	0	54	50.08
128	2*2*2*2*2*2*2	688	133.7k	431.4k	0	50	88.06

the radix-2 and radix-4 variants. As anticipated, the radix-4 variant uses 21% fewer DSP slices with respect to the radix-2 one, and achieves comparable performance. This is the largest implementation, and occupies 10% of the total LUTs, 14% of the FFs and 13% of all the DSPs.

B. LUT-based FFT implementations

The designs in this section are implemented without using the dedicated multipliers available in the DSP blocks. This approach is more appropriate for ASIC designs, as well as for some families of FPGAs, since it does not rely on hardened IPs for their realisation. These designs prove that the proposed architecture is efficient in terms of area without compromising performance. The developed architecture allows for fine grain control of the pipeline registers, leading to reduce the critical paths while maintaining the synchronization of the data.

The designs are implemented using 16 bits for the real and the imaginary parts, noted (16+16). However, the proposed IP core is parameterised, and can be configured with arbitrarily large or small signals, as well as with custom fixed-point number representations. The reference systems shown use 5 bits for the integer part and 11 bits for the fractional one. This codification has been measured to avoid overflow in the integer part. However, it can be adapted depending on the magnitude of the signals to be processed.

The trends in resource utilization follow the expected patterns. As the number of points processed grows, the number of used LUTs and FFs also increases. Similarly, power-of-two FFTs require the fewest resources while power-of-three decompositions are less efficient. Smaller systems operate at a theoretical maximum of 891 MHz, while there is a slight drop-off in performance on larger systems. These systems would require further optimisation to achieve the highest clock rates.

C. Resource comparison and overall performance

Comparing Table I and Table II, we see that the DSP-based implementations obtain higher clock rates and slightly higher SQNRs due to the fact they are (18+18)-bit wide vs. (16+16)-bit wide. The cost to pay is the usage of some DSP multipliers for multiplications by constants, which may seem wasteful for this resource. The LUT-based approach saves all the DSP slices, but it obviously occupies more LUTs and also more registers. For example, the 64-point LUT-based implementation uses 42.3% more LUTs, 50.1% more registers

but 377 (100%) fewer DSP multipliers. From the 16- and 32-point circuits we see how different decompositions have little impact on the final result.

The reference parallel radix-2 and radix-4 architectures have been extracted from [23], but in all cases we have obtained faster implementations, in part due of the upgrade to the Virtex Ultrascale+ platform from Virtex Ultrascale. In the 32-point FFTs, we increase throughput by 40% in both of our proposed architectures. The DSP-based version does so occupying the same number of LUTs, 106 DSP slices and some extra registers. The LUT-based architecture occupies 69% more LUTs compared to the reference one. In the 64-point FFTs, the DSP-based block achieves a throughput 54% higher, LUT usage remains the same and 377 DSP multipliers are used. In the 128-point FFTs, by occupying additional 790 DSP blocks we obtain the results 47% faster in terms of throughput, netting us 100 GS/s.

V. Conclusions

In this paper we have demonstrated the effectiveness of our methodology for the generation of power-of-two and non-power-of-two FFT implementations. Our approach is highly versatile and capable of providing very high performance architectures for the required number of points. The ease of definition, paired with the convenience of pipelining, results in a wide array of very high performance FFT implementations.

We have measured the efficiency of utilization of the resources as a function of the applied factorisation in each case. We have summarised the mathematical theory behind the Fourier Integral Theorem. We have deduced the expressions of the FFT from the Discrete Fourier Transform, and discussed the matrix form of the FT and the FFT decompositions. Finally, we have generated and implemented the RTL descriptions of a number of very high performance FFT realisations with different number of points and types of factorisations.

References

[1] G. Akkad, A. Mansour, B. ElHassan, F. Le Roy, and M. Najem, "FFT radix-2 and radix-4 FPGA acceleration techniques using HLS and HDL for Digital Communication Systems," in *2018 IEEE Int. Multidisc. Conf. Eng. Tech. (IMCET)*. IEEE, 2018, pp. 1–5.

[2] I. S. Uzun, A. Amira, and A. Bouridane, "FPGA implementations of fast Fourier transforms for real-time signal and image processing," *IEEE Proc. Vis., Image, Sign. Proc.*, vol. 152, no. 3, pp. 283–296, 2005.

[3] I. Hatai, R. Biswas, and S. Banerjee, "ASIC implementation of a 512-point FFT/IFFT processor for 2D CT image reconstruction algorithm," in *IEEE Tech. Students' Symp.* IEEE, 2011, pp. 220–225.

[4] A. Černỳ, "Introduction to fast Fourier transform in financeS," *The Journal of Derivatives*, vol. 12, no. 1, pp. 73–88, 2004.

[5] R. Buccheri and B. Sacco, "Time analysis in astronomy: Tools for periodicity searches," in *Data Analysis in Astronomy*. Springer, 1985, pp. 15–27.

[6] Y. Lu, Y. Huang, W. Xue, and G. Zhang, "Seismic data processing method based on wavelet transform for denoising," *Cluster Computing*, vol. 22, no. 3, pp. 6609–6620, 2019.

[7] M. Frigo and S. G. Johnson, "FFTW: Fastest Fourier transform in the west," *Astrophysics Source Code Library*, pp. ascl–1201, 2012.

[8] NVIDIA, "CUFFT Library," May 2020. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html

[9] N. Corporation, "NVIDIA CUDA Toolkit," May 2020. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[10] E. Wold and A. Despain, "Pipeline and parallel-pipeline FFT processors for VLSI implementations," *IEEE Trans. Comp.*, vol. 33, no. 05, pp. 414–426, 1984.

[11] C.-W. J. W.-C. Yeh, "High-speed and low-power split-radix FFT," *IEEE Trans. Sign. Proc.*, vol. 51, no. 3, pp. 864–874, 2003.

[12] D. Poole, *Linear algebra: A modern introduction*. Cengage Learning, 2014.

[13] M. Gasca and T. Sauer, "On the history of multivariate polynomial interpolation," in *Numerical Analysis: Historical Developments in the 20th Century*. Elsevier, 2001, pp. 135–147.

[14] W. Fulton, *Algebraic curves: an introduction to algebraic geometry*. Addison-Wesley, 1989.

[15] R. P. Feynman, R. B. Leighton, and M. Sands, "The Feynman lectures on physics; Vol. I," *American Journal of Physics*, vol. 33, no. 9, pp. 750–752, 1965.

[16] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical analysis*. Cengage learning, 2015.

[17] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal of the ACM (JACM)*, vol. 15, no. 2, pp. 252–264, 1968.

[18] H. Sloate, "Matrix representations for sorting and the fast Fourier transform," *IEEE Trans. Circ. Syst.*, vol. 21, no. 1, pp. 109–116, 1974.

[19] A. Cortés, I. Vélez, and J. F. Sevillano, "Radix r^k FFTs: Matricial representation and SDC/SDF pipeline implementation," *IEEE Trans. Sign. Proc.*, vol. 57, no. 7, pp. 2824–2839, 2009.

[20] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[21] H. Prodinger, "On binary representations of integers with digits -1, 0, 1," *Integers*, pp. A8–14, 2000.

[22] C. R. Harris *et al.*, "Array programming with numpy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020.

[23] M. Garrido, K. Möller, and M. Kumm, "World's fastest FFT architectures: Breaking the barrier of 100 GS/s," *IEEE Trans. Circ. Syst. I: Reg. Papers*, vol. 66, no. 4, pp. 1507–1516, 2018.