Towards a Flexible Hardware Implementation for Mixed-Radix Fourier Transforms

Mario Vega, Xiaokun Yang, John Shalf, Doru Thom Popovici
Lawrence Berkeley National Lab
{mvega, xiaokunyang, jshalf, dtpopovici}@lbl.gov

***Abstract*—The discrete Fourier transform is a versatile mathematical kernel widely used in a myriad of applications from physics, chemistry, and even machine learning. While most applications typically rely on power of two sizes for which high performance implementations have been developed as software libraries or custom hardware IPs, there are codes in chemistry or even machine learning that require non-power of two or even prime-sized Fourier transforms. Classic prime-sized Fourier transform implementations are more complicated and less performant both in software and in hardware compared to the power of two implementations. Recent work [1] has shown that casting small prime-sized Fourier transforms as specialized matrix operations, competitive codelets can be developed on CPU platforms. In this work, we focus on designing the corresponding hardware unit for prime-sized Fourier transforms and integrating the custom design within any mixed-radix Fourier transform. We provide a detailed analysis of the design and investigate the latency, throughput, and resource utilization on Xilinx FPGAs for the both standalone and mixed-radix designs. We show that even oddly shaped Fourier transforms can be appealing when building custom hardware designs for general size Fourier transforms.**

***Index Terms*—Fourier transforms, mixed-radix, prime-sizes, specialized matrix operation, FPGA**

I. Introduction

The discrete Fourier transform (DFT) is a key mathematical tool that appears in a wide range of applications from fields like physics, chemistry, material sciences and even machine learning. Typically, most applications require power of two sizes for which high performance software libraries and hardware IPs have been developed and studied over the past years. However, there are scenarios, where codes require non-power of two sizes or even prime-sized Fourier transforms. For example, DFT-based pseudo-spectral solvers seen in particle in cell (PIC) codes [2], DFT-based eigensolvers [3], [4] used in planewave Density Functional Theory codes, or DFT-based convolutions used in Local Correction (MLC) codes [5] and even machine learning applications [6]–[8] may require oddly shaped Fourier transforms. For such sizes, the typical Cooley-Tukey algorithm [9], meant for power of two sizes, cannot be directly utilized off the shelf.

Typically, the mixed-radix Cooley-Tukey algorithm decomposes any composite number into smaller building blocks, that are implemented as codelets for small powers of two and small prime sizes. The FFTW library [10] follows this approach, where a multitude of codelets are created and then linked together to create any composite size. For power of two sizes, codelets of size 2, 4, 8 can be used. For general composite sizes, prime sizes are needed, for which algorithms like Rader [11] or Bluestein [12] have been developed. Both cast the Fourier computation as a circular convolution by either permuting the data or padding the original data and increasing the problem size by at least twice the original size. Both algorithms exhibit worse execution time as compared to the power of two counterpart. Recent work by Popovici et. al [1] has shown that the Fourier computation for small prime sizes can be cast as specialized matrix operations that exploit the structure of the Fourier computation. This approach has been shown to outperform classical implementations within FFTW on CPUs.

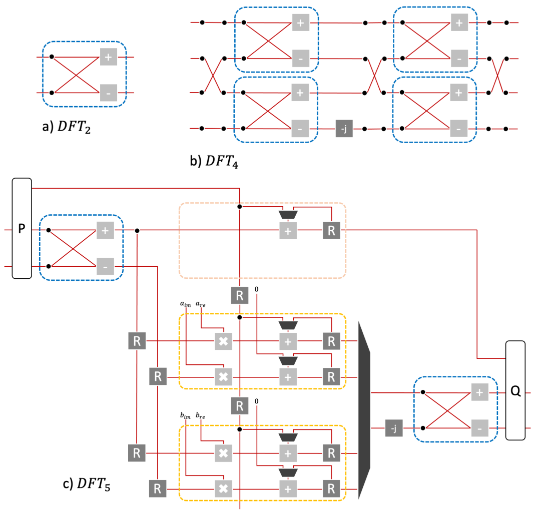
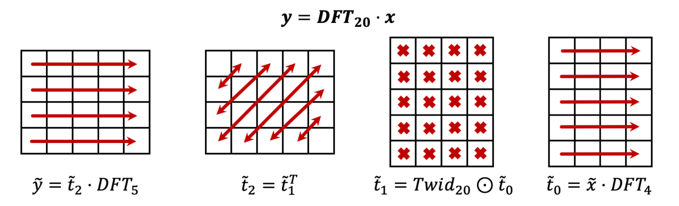


Fig. 1: The hardware design for a DFT of size a) 2, b) 4 and c) $p = 5$. These building blocks are sufficient for building any Fourier transform of size 2^k5^l.

The idea that any composite size Fourier transform only needs a handful of codelets as opposed to FFTW's approach of having a large number of codelets, becomes appealing. This aspect is of great interest nowadays with hardware specialization. In this work, we focus on developing a specialized prime-sized Fourier transform and integrating the IP within any mixed-radix Fourier transform. As depicted for any Fourier transform of size 2^k5^l in Figure 1, we plan to utilize a reduced set of building blocks. We provide details about the hardware design choices and outline the trade-offs between latency, throughput, and resource utilization on Xilinx FPGAs. We



$y = DFT_{20} \cdot x$
$\tilde{y} = \tilde{t}_2 \cdot DFT_5$
$\tilde{t}_2 = \tilde{t}_1^T$
$\tilde{t}_1 = Twid_{20} \odot \tilde{t}_0$
$\tilde{t}_0 = \tilde{x} \cdot DFT_4$

Fig. 2: The decomposition of a DFT of size 20 using a mixed-radix algorithm. The 1D DFT is viewed as a set of linear algebra operations, where the 1D input and output x and y are viewed as 2D matrices $\tilde{x}$ and $\tilde{y}$. The decomposition applies DFTs of size 5 and 4 on the data, a point-wise scaling with the twiddle factors, and a transposition step.

outline the different characteristics of the integrated designs. Finally, we will provide an end-to-end implementation by developing a lightweight code generator, using the Chisel Hardware Construction Language (HCL), to allow for fast prototyping of the designs on any FPGA systems.

Contributions. The paper makes the following contributions:

1) We develop a custom IP for prime-sized Fourier transforms and perform a detailed analysis of its performance characteristics.
2) We integrate the custom design within a mixed-radix implementation, analyzing the trade-offs between latency, throughput, and resource utilization on modern FPGA systems.
3) We provide an end-to-end tool implemented in the Chisel language to enable ease of prototyping for any composite size Fourier transform.

II. Background

In this section, we briefly present the Fourier transform. We focus on the mixed-radix implementation using both power of two and prime-sized kernels. We outline the details for the prime-sized transforms in [1].

A. The Fourier Transform

The discrete Fourier transform is a linear transform such as

$$y = DFT_N \cdot x, \tag{1}$$

where x/y represent one dimensional input/output arrays of size N. The DFT_N represents the complex type Fourier matrix defined as

$$DFT_N = \left[\omega_N^{kl}\right], \forall\ 0 \leq k < N \text{ and } 0 \leq l < N, \tag{2}$$

where $\omega_N = e^{-j\frac{2\pi}{N}}$ and $j^2 = -1$. The elements of the Fourier matrix represent the roots of unity for a given size N.

The Fourier transform for composite sizes $N = nm$ is implemented using fast algorithms like the Cooley-Tukey algorithm [9], that factorizes the Fourier matrix. For example, the Fourier matrix in Equation 1 is re-written as a set of linear algebra operations such as

$$\tilde{y} = \left(Twid_{m\times n} \odot (\tilde{x} \cdot DFT_n)\right)^T \cdot DFT_m, \tag{3}$$

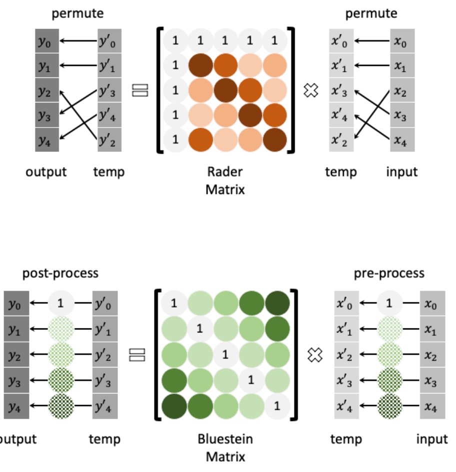


Fig. 3: The Rader and Bluestein algorithms used for implementing prime-sized Fourier transforms. Both algorithms require pre/post processing steps to reshape the matrix into a circulant matrix. The circulant matrix is then decomposed into a forward, inverse Fourier transform and a diagonal matrix.

where $\tilde{x}/\tilde{y}$ represent the 2D matrix views of the original input/output x/y. The input $\tilde{x}$ is an $m \times n$ matrix, and the output $\tilde{y}$ is an $n \times m$ matrix, both stored in column major order. The first step of the decomposition applies a DFT of size n in the rows of $\tilde{x}$. The result is then point-wise scaled with the so-called twiddle factors $Twid_{m\times n}$, and then transposed from an $m \times n$ matrix to an $n \times m$ matrix. A DFT of size m is applied in the rows of the transposed matrix to obtain the final output $\tilde{y}$. The decomposition is depicted in Figure 2. For power of two sizes, the decomposition will always deal with smaller power of two sizes, the smallest being the DFT of size 2 defined as the butterfly matrix

$$DFT_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{4}$$

For composite sizes like $N = 20$, the decomposition requires a DFT of size 4 and a DFT of size 5, for which different algorithms must be used.

Typically, Rader's [11] or Bluestein's [12] algorithms can be used to compute prime-sized DFTs. As depicted in Figure 3, both algorithms restructure the Fourier matrix as a circulant matrix (a matrix where each column is a shifted version of the previous column). The circulant matrix is the decomposed as a forward, inverse Fourier transform and a diagonal matrix. Frameworks like Spiral [13] utilize the two algorithms to generate software or hardware for such problem sizes. The work by Milder et. al [14], [15] has outlined that prime-sized Fourier transforms and indirectly mixed-radix DFTs are more expensive in execution time and resource utilization, compared to the power of two counterparts. Recent work [1] has shown that a simpler implementation can be obtained, which will be presented in the following sub-section.

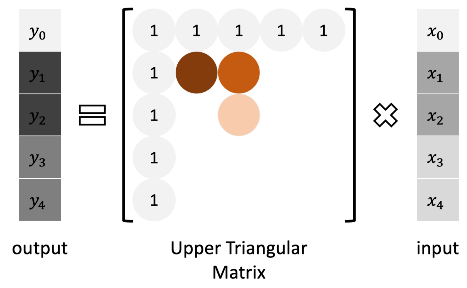


Fig. 4: The implementation of a DFT of size 5, where only the columns and rows of 1 and the upper triangular matrix are needed to fully compute the Fourier transform. More details can be found in [1].

B. *The Prime-Sized DFT Kernel*

The work by Popovici et. al [1] has outlined an alternative approach for computing prime-sized Fourier transforms. Instead of reshaping the Fourier matrix, the paper exploits the structure of the matrix. For consistency, we will provide a brief description of the approach and refer the readers to the paper for more details. As an example, we use $y = DFT_5 \cdot x$ which can be expanded as

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} \omega_5^0 & \omega_5^0 & \omega_5^0 & \omega_5^0 & \omega_5^0 \\ \omega_5^0 & \omega_5^1 & \omega_5^2 & \omega_5^3 & \omega_5^4 \\ \omega_5^0 & \omega_5^2 & \omega_5^4 & \omega_5^6 & \omega_5^8 \\ \omega_5^0 & \omega_5^3 & \omega_5^6 & \omega_5^9 & \omega_5^{12} \\ \omega_5^0 & \omega_5^4 & \omega_5^8 & \omega_5^{12} & \omega_5^{16} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}. \tag{5}$$

Using the properties that $\omega_5^0 = 1$ and $\omega_5^k = \omega_5^{k-5}$, the Fourier matrix can be re-written as

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_5 & \omega_5^2 & \omega_5^{-2} & \omega_5^{-1} \\ 1 & \omega_5^2 & \omega_5^{-1} & \omega_5 & \omega_5^{-2} \\ 1 & \omega_5^{-2} & \omega_5 & \omega_5^{-1} & \omega_5^2 \\ 1 & \omega_5^{-1} & \omega_5^{-2} & \omega_5^2 & \omega_5^1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}. \tag{6}$$

The first element in the output vector y is computed as

$$\begin{aligned} y_0 &= x_0 + 1 \cdot x_1 + 1 \cdot x_2 + 1 \cdot x_3 + 1 \cdot x_4, && (7) \\ &= x_0 + 1 \cdot (x_1 + x_4) + 1 \cdot (x_2 + x_3) && (8) \end{aligned}$$

The grouping of the input terms in the above formula will be clarified in the following paragraphs. The computation of the remaining outputs can be done in pairs. For example, the pair (y_1, y_4) can be computed as

$$\begin{aligned} y_1 &= x_0 + \omega_5 \cdot x_1 + \omega_5^2 \cdot x_2 + \omega_5^{-2} \cdot x_3 + \omega_5^{-1} \cdot x_4 && (9) \\ y_4 &= x_0 + \omega_5^{-1} \cdot x_1 + \omega_5^{-2} \cdot x_2 + \omega_5^2 \cdot x_3 + \omega_5 \cdot x_4, && (10) \end{aligned}$$

where ω_5^{-1} and ω_5^{-2} represent the complex conjugates of ω_5 and ω_5^2, respectively. The paper exploits the complex conjugate property of the ω terms by grouping the input terms such as

$$\begin{aligned} &\omega_5 \cdot x_1 + \omega_5^{-1} \cdot x_4 && (11) \\ &\omega_5^{-1} \cdot x_1 + \omega_5 \cdot x_4. && (12) \end{aligned}$$

The ω terms can be expanded as real and imaginary parts, and the computation can further be grouped as

$$\begin{aligned} &Re\{\omega_5\} \cdot (x_1 + x_4) + j \cdot Im\{\omega_5\} \cdot (x_1 - x_4) && (13) \\ &Re\{\omega_5\} \cdot (x_1 + x_4) - j \cdot Im\{\omega_5\} \cdot (x_1 - x_4), && (14) \end{aligned}$$

where $Re\{\cdot\}$ and $Im\{\cdot\}$ represent the real and imaginary parts of a complex number. Utilizing the same principle for the x_2 and x_3, the overall computation of the (y_1, y_4) output pairs can be modified as

$$\begin{aligned} t_r &= x_0 && (15) \\ t_i &= 0 && (16) \\ t_r &+= Re\{\omega_5\} \cdot (x_1 + x_4) && (17) \\ t_i &+= Im\{\omega_5\} \cdot (x_1 - x_4) && (18) \\ t_r &+= Re\{\omega_5^2\} \cdot (x_2 + x_3) && (19) \\ t_i &+= Im\{\omega_5^2\} \cdot (x_2 - x_3) && (20) \\ y_1 &= t_r + j * t_i && (21) \\ y_4 &= t_r - j * t_i. && (22) \end{aligned}$$

Similarly, the output pair (y_2, y_3) can also be computed.

The overall computation is simplified. The complex multiplication with the twiddle factors is replaced with real multiplication. The real and imaginary parts of the ω terms are multiplied against pairs of input terms, i.e. (x_1, x_4) and (x_2, x_3). An astute reader may identify the DFTs of size 2 being used between the pairs of inputs and outputs, respectively. As depicted in Figure 1, this implementation can easily be implemented in hardware.

III. Building the Hardware Components

In this section, we present our approach to generating hardware designs for prime-sized and mixed-radix Fourier transforms. In our implementations we aim at generating streaming DFT designs, where data flows through the components continuously. For the power of two sizes, bit-wise permutations and point-wise multiplications with the twiddle factors are mapped as streaming sub-modules and pieced together with DFTs of sizes 2 and 4, following the implementations provided by Spiral [13].

A. *Designing the Prime-Sized Fourier Kernel*

We use Chisel Hardware Construction Language (HCL) [16] to construct a template for the prime-sized Fourier transform as depicted in Figure 5. The template has a top module that defines the prime-sized Fourier transform. The top module is built up from five sub-modules, i.e. two data adjustment sub-modules, two modules that perform pair-wise addition and subtractions (DFT_2s), and one multiply and accumulate (MAC) module. The manner in which the sub-modules are instantiated and pieced together is determined by 1) the parameter configurations in the top modules and 2) the patterns baked into the top module. The order of the sub-modules does not change, only the stitching between each module. For the prime DFT generator, the parameters include prime-DFT size, streaming width, and floating point precision. In the following

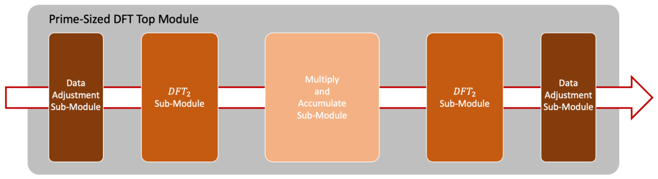

Prime-Sized DFT Top Module
Data Adjustment Sub-Module
DFT_2 Sub-Module
Multiply and Accumulate Sub-Module
DFT_2 Sub-Module
Data Adjustment Sub-Module

Fig. 5: The Chisel template for the prime-sized Fourier transform. The top module contains the corresponding modules for data permutation, pair-wise addition and subtraction, and multiply and accumulate. The top module specifies the size of the DFT, the streaming width, and the floating point precision.

paragraphs, we will discuss the data adjustment and MAC sub-modules. The DFT_2 sub-modules are just butterfly operators as depicted in Figure 1.

Data Adjustment Sub-Modules. The sub-modules reorganize the data such that the data points are in the correct order. For example, for the implementation of a DFT_5, the input adjustment sub-module extracts the first element x_0, and forms the pairs (x_1, x_4) and (x_2, x_3), while the output adjustment module shuffles y_0 and pairs (y_1, y_4) and (y_2, y_3) back to the original order. The extract and shuffle operations can be seen in Figure 6. In general, the prime-sized Fourier transform reads input values x_i and x_{n-i} to produce the output values y_i and y_{n-i}, for all values $i = 1..(n-1)/2$. Additionally, the first element of the input x_0 and the first element of the output y_0 need to be accumulated separately. The Chisel implementation of these sub-modules follows the general details from the work by Püschel et. al [17], where permute units are implemented as switch-memory-switch blocks.

Multiply and Accumulate Sub-Module. The most time consuming part for any prime-sized Fourier transform is represented by the multiply and accumulate (MAC) operations as outlined in Equation 8 and Equations 17-20. The equations can be translated into an array of MAC units as depicted in Figure 7, where we show the array of MACs for the DFT_5 example. The top adder accumulates the updates to y_0, while the bottom two groups of multipliers and adders perform the corresponding updates to the rest of outputs. The first group updates the output pairs (y_1, y_4), and the second updates (y_2, y_3). The three groups work in a pipeline fashion.

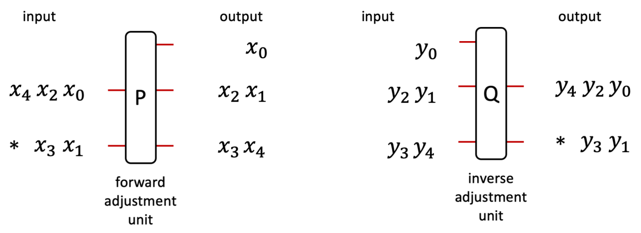


Fig. 6: The P and Q units manipulate the data. The computation required for any prime-sized Fourier transform needs the data to be shuffled and reorganized. P extracts the first element x_0 and creates the pairs (x_1, x_4) and (x_2, x_3). The Q block performs the opposite.

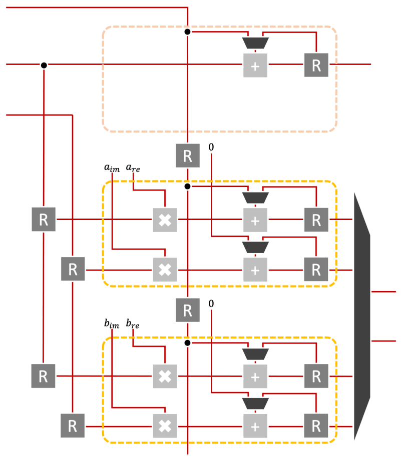


Fig. 7: The diagram that outlines the accumulator and multiply and accumulate (MAC) units required by the DFT_5. The accumulator performs the updates to the first term y_0 of the output. The units work in a systolic fashion, each group working on different pairs of data at a time (note the registers R between the three groups).

We use registers in between the groups of operations to delay the data between the units. In the first iteration, two complex numbers produced by the DFT_2 are passed to the accumulator that updates y_0. The same two complex numbers are stored into registers to be subsequently used in the next iteration. In the next iteration, while the accumulator unit updates y_0 with the new values obtained from the DFT_2, the first MAC unit operates on the previously stored complex numbers. Each complex number is multiplied with the corresponding real and imaginary components of the roots of unity and accumulated to the corresponding registers. In each iteration, new data points are produced by the DFT_2 until all blocks are processing in parallel. Once the iterations have managed to stream the data through, each group will output the results to a second DFT_2 to form the final output results.

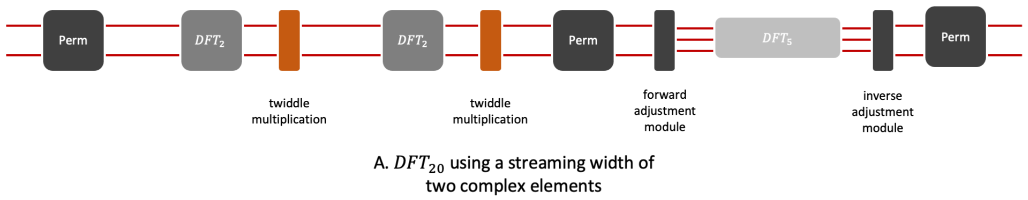
Note that the DFT_5 requires an accumulator to hold the updates to y_0 and two groups MAC units for the remaining four output elements. In general, for any prime-sized Fourier transform, the total number of MAC groups, including the accumulator, can be determined as

$$n_{MAC} = 1 + (p-1)/2, \tag{23}$$

where p represents the prime size. Using this formula we can estimate the total number of floating point adders and multipliers needed by our implementation. Therefore, the new prime-sized Fourier transform requires $2 * (p-1) + 2$ adders and $2 * (p-1)$ multipliers. In addition, a constant amount of eight adders is needed to account for the floating point adders present in the two DFT_2.

B. The Mixed-Radix Fourier Transform

Similar to the prime-sized Fourier transform, we create a templatized mixed-radix Fourier transform using the Chisel


Perm
DFT_2
DFT_2
Perm
DFT_5
Perm
twiddle multiplication
twiddle multiplication
forward adjustment module
inverse adjustment module
A. DFT_{20} using a streaming width of two complex elements

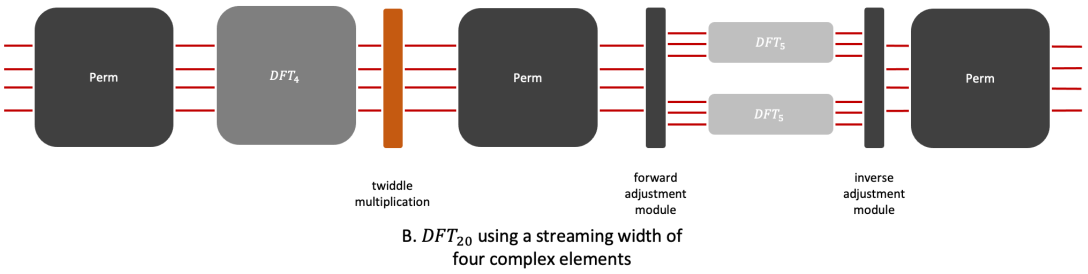

Perm
DFT_4
Perm
DFT_5
DFT_5
Perm
twiddle multiplication
forward adjustment module
inverse adjustment module
B. DFT_{20} using a streaming width of four complex elements

Fig. 8: Two designs that correspond to the mixed-radix DFT_{20}. Design A. has an overall streaming width of two complex elements per cycle. As building blocks the design uses DFT_2s, a DFT_5, some point-wise scaling operations (orange boxes) and some permutation units (dark grey boxes). Design B. has an overall streaming width of four complex elements per cycle. The computation is decomposed as DFT_4 and DFT_5. The streaming width changes the permutation units.

Size	Prime-Sized Kernel	Bluestein to closest 2^n
3	14 adds, 4 muls	44 adds, 40 muls
5	18 adds, 8 muls	60 adds, 48 muls
7	22 adds, 12 muls	60 adds, 48 muls
11	30 adds, 20 muls	76 adds, 54 muls
13	34 adds, 24 muls	76 adds, 54 muls

TABLE I: Comparing the number of floating point operations for our approach of implementing the prime-sized Fourier transform against the Bluestein algorithm. The Bluestein approach decomposes the circular matrix outlined in Figure 3 using a forward and inverse DFT and a point-wise scaling operation of size greater than $2p$, where p is the prime size. Usually the Bluestein algorithm increases the problem size to a power of two for which fast DFT algorithms exist.

language. We define a top module as the main transform and the corresponding sub-modules for the different components. The sub-modules are represented by the DFTs of the different radix, permutation units, and point-wise scaling operations. For the DFTs, we restrict our approach only to DFTs of size two and four and prime size p. In addition, we organize the DFT sub-modules such that the power of two sub-modules go first, and then the prime-sized transforms. Depending on the decomposition, the top-module instantiates and links the corresponding sub-modules together. The top module accepts as parameters the size of the Fourier transform, the streaming width, and the data type. The configuration of parameters has similar implications to the stand-alone prime DFT in terms of influencing the hardware cost, parallelism, and precision.

Figure 8, outlines two hardware designs for the mixed-radix DFT_{20}. The first design imposes a streaming width of two complex elements per cycle. For this design, the Chisel generator chooses the DFT_2 and DFT_5 as the main compute units and configures the permutation units accordingly. The second design is configured with an increased streaming width of four complex numbers per cycle. This in turn modifies the computation kernels, by choosing a DFT_4 instead of two DFT_2s. The permutation units are also configured to allow four elements as input and four elements as output. Note that for the this implementation, there are two DFT_5s. These two implementations are needed to remove any bubbles in the pipeline, which may be introduced given the extra zeros for the prime-sized Fourier transform. All other mixed-radix Fourier transforms are generated in a similar fashion. In the following section, we present some results using our approach for generating mixed-radix Fourier transforms.

IV. Experimental Results and Discussion

In this section, we present experimental results for the generated hardware for both the prime-sized Fourier transform and the mixed-radix implementation. For each experiment, we analyze our approach and compare it against other hardware implementations, when possible. We also provide a discussion of current limitations and future directions.

Methodology. As stated in the previous section, we have created templatized implementations for both the prime-sized Fourier transform and mixed-radix implementation, using the Chisel HCL. The current templates are meant to generate streaming architectures, where data flows from one sub-module to another. We allow the user to specify the Fourier transform size, the streaming width and the data type. The tool will instantiate the sub-modules, create the temporary buffers and link everything together. The generated Verilog code is then passed through the Vivado 2023 tool chain to synthesize

Size	Latency [cycles]	LUTs	FFs	Memory [kB]
3	81	10,253	14,493	0.9
5	124	14,361	20,446	1.8
7	167	18,532	26,066	2.7
11	253	36,035	37,487	4.5
13	296	39,755	43,281	5.4

TABLE II: Results for prime-sized Fourier transforms. We report the latency and the resource utilization when synthesizing the hardware for the FPGA.

and do "place and route" for the design on the U280 Xilinx FPGA board.

Prime-Sized Fourier Transforms. We focus first on the prime-sized Fourier transform. Neither Spiral, nor the Xilinx IP generator provide readily available hardware for prime-sizes. From previous papers [14], [15], we can extract that prime-sizes are implemented using the Bluestein algorithm. As such, we provide a comparison of floating point operation count between our approach and the Bluestein algorithm, and outline the resource utilization of our approach when targeting the U280 Xilinx FPGA. Table I outlines the different number of floating point operations needed by our approach and the classical implementation. The Bluestein algorithm applies a pre/post processing step to convert the Fourier matrix into a circular matrix as outlined in Figure 3. The matrix is not directly applied on the data, rather it is decomposed into a forward DFT, a point-wise scaling operation and an inverse Fourier transform. The size of the Fourier transforms is at least twice the size of the original prime size p. Most implementations choose to increase the problem size to the nearest power of two, for which efficient implementations of the DFTs have already been developed. The problem size increase requires data padding with zeros, which may cause bubbles in the hardware pipelined design. In contrast, our approach requires simple shuffle operations, and a computation that can be done with a matrix multiply engine. Table II outlines the resource utilization as estimated by the Vivado tool chain. We report the number of LUTs, FFs and BRAM utilization. We also report the latency of each of design. For all the designs, the Vivado tool chain estimates a frequency of 280 MHz on a Xilinx Alveo U280 board.

Mixed-Radix Algorithm for Composite Sizes. We integrate the prime-sized Fourier transform into the mixed-radix implementation. We explore different design choices for composite sizes like 20, 28, 88, 96 and 192. Table III outlines the latency and the resource utilization for each of the designs. Note that we generate different designs based on the overall streaming width (the input and output streaming width of the entire DFT). When we increase the streaming width, our approach increases the power of two sizes from two to four and also duplicates the prime-sized units. Recall that the prime-sized Fourier transform requires an insertion of a zero term, which may cause pipeline bubbles. As expected, increasing the streaming width, will require more resources to be utilized. However, the number of stages is reduced and hence the latency is decreased. All the tested designs were able to fit on the U280 Xilinx FPGA board.

Size	SW	Latency (cycles)	LUTs	FFs	Memory [kB]
20	2	436	28,630	38,454	9.5
20	4	357	57,796	67,530	13.6
28	2	562	33,782	44,124	13.4
28	4	469	67,441	78,954	19.7
88	2	1,027	50,764	69,829	27.8
88	4	857	104,413	130,802	37.2
96	2	901	66,316	105,946	19.4
96	8	439	187,674	273,283	26.1
192	2	1,532	101,699	180,964	36.8
192	8	692	260,575	375,404	43.5

TABLE III: Results for the mixed-radix implementations using our prime-sized Fourier transform. We report the latency and the resource utilization when synthesizing the hardware on the U280 FPGA.

Discussion. We focused on the specialized implementation for prime-sized and mixed-radix Fourier transforms. We presented some preliminary results. We could not compare the designs with Spiral or the Xilinx IP generator, since neither frameworks offered readily available hardware designs for the selected sizes. We compared the power of two sizes generated by our tool against the designs generated by Spiral generated ones, and the designs are competitive. There are still optimizations that need to be added to our approach. First, the current implementation of the prime-sized Fourier transform is restricted to a streaming width of two complex elements (input and output). Increasing the streaming width requires some minor modifications to the array of multiply and accumulate units. Secondly, our designs require more buffers. The adjustment sub-modules require buffers to reshape the data. However, for mixed-radix implementations, the permutation units and the adjustment modules can be merged into a single sub-module to reduce the required memory. Lastly, the current generator creates one dimensional Fourier transforms for batched and non-batched data. However, the units must be used as part of two dimensional and three dimensional Fourier computations needed by planewave DFT codes. All these we leave as future work.

V. Conclusion

This paper presents a new hardware solution for prime-sized and mixed-radix Fourier transforms. We have implemented an alternate kernel for prime-sizes, following the details from a previous work. We have incorporated the novel kernel design into a lightweight generator constructed with the Chisel Hardware Language, which enables us to generate Verilog code for various Fourier transforms. We used the generator to experiment with Fourier transforms of different sizes and streaming widths. Overall, we have provided a solution for composite sizes that require fewer number of computational codelets/kernels compared to the FFTW library, hence making forward steps in providing a flexible hardware implementation for general sizes Fourier transforms.

Acknowledgement

First, we would like to thank the reviewers for the valuable comments that helped improve and make the paper stronger. Second, we would also like to thank Patricia Gonzalez-Guerrero for the feedback and directions to make the story more concise. Finally, this manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

References

[1] D. T. Popovici, D. N. Parikh, D. G. Spampinato, and T. M. Low, "Exploiting symmetries of small prime-sized dfts," in *Parallel Processing and Applied Mathematics: 13th International Conference, PPAM 2019, Bialystok, Poland, September 8–11, 2019, Revised Selected Papers, Part I 13*. Springer, 2020, pp. 162–173.

[2] J.-L. Vay, A. Almgren, J. Bell, L. Ge, D. Grote, M. Hogan, O. Kononenko, R. Lehe, A. Myers, C. Ng *et al.*, "Warp-X: A new exascale computing platform for beam–plasma simulations," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 2018.

[3] C.-K. Skylaris, P. D. Haynes, A. A. Mostofi, and M. C. Payne, "Introducing ONETEP: Linear-scaling density functional simulations on parallel computers," *The Journal of Chemical Physics*, vol. 122, no. 8, p. 084119, 2005.

[4] D. T. Popovici, F. P. Russell, K. Wilkinson, C.-K. Skylaris, P. H. Kelly, and F. Franchetti, "Generating optimized Fourier interpolation routines for density functional theory using SPIRAL," in *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2015, pp. 743–752.

[5] P. McCorquodale, P. Colella, G. Balls, and S. Baden, "A local corrections algorithm for solving poisson's equation in three dimensions," *Communications in Applied Mathematics and Computational Science*, vol. 2, no. 1, pp. 57–81, 2007.

[6] A. Zlateski, Z. Jia, K. Li, and F. Durand, "The anatomy of efficient fft and winograd convolutions on modern cpus," in *Proceedings of the ACM International Conference on Supercomputing*, 2019, pp. 414–424.

[7] A. Zlateski, Z. Jia, K. Li, and Durand, "Fft convolutions are faster than winograd on modern cpus, here is why," 2018.

[8] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with fbfft: A GPU performance evaluation," *arXiv preprint arXiv:1412.7580*, 2014.

[9] 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.

[10] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 216–231, 2005.

[11] C. M. Rader, "Discrete Fourier transforms when the number of data samples is prime," *Proceedings of the IEEE*, vol. 56, no. 6, pp. 1107–1108, 1968.

[12] L. Bluestein, "A linear filtering approach to the computation of discrete Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 4, pp. 451–455, 1970.

[13] F. Franchetti, T. M. Low, D. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, "SPIRAL: extreme performance portability," *Proc. IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018. [Online]. Available: https://doi.org/10.1109/JPROC.2018.2873289

[14] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Hardware implementation of the discrete fourier transform with non-power-of-two problem size," in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2010, pp. 1546–1549.

[15] P. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Computer generation of hardware for linear digital signal processing transforms," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 17, no. 2, pp. 1–33, 2012.

[16] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1216–1225.

[17] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using rams," *Journal of the ACM (JACM)*, vol. 56, no. 2, pp. 1–34, 2009.