HARDWARE IMPLEMENTATION OF THE DISCRETE FOURIER TRANSFORM WITH NON-POWER-OF-TWO PROBLEM SIZE

Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel

Carnegie Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA, United States

ABSTRACT

In this paper, we examine several algorithms suitable for the hardware implementation of the discrete Fourier transform (DFT) with non-power-of two problem size. We incorporate these algorithms into Spiral, a tool capable of automatically generating corresponding hardware implementations. We discuss how each algorithm can be used to generate different types of hardware structures, and we demonstrate that our tool is able to produce hardware implementations of non-power-of-two sized DFTs over a wide range of cost/performance tradeoff points.

Index Terms— Discrete Fourier transform, high-level synthesis, field-programmable gate array, FPGA, algorithms

1. INTRODUCTION

Transforms such as the discrete Fourier transform (DFT) are very commonly used in signal processing, communication, and scientific computing applications. However, they are difficult and time-consuming to implement in hardware because of the large number of algorithmic and datapath options that must be considered in order to tailor the implementation to a particular application's cost and performance requirements. The right choices are highly dependent on the context. These problems considerably worsen when the transform size is not a power of two: algorithms become less regular and the relationship between algorithm and hardware implementation becomes more complicated. For these reasons, most existing work on hardware implementation of the DFT with non-power-of-two problem size focuses on building an implementation to meet a specific application's cost and performance goals, and not on systematically creating a space of designs with different cost/performance characteristics.

Contribution. In this paper we utilize the hardware generation framework underlying Spiral [1, 2] to automatically produce hardware implementations of the DFT with non-power-of-two problem size. We consider several algorithmic options including the Bluestein [3] and mixed radix [4] algorithms, and we examine their applicability to the mathematical framework we use in hardware generation. Because designs are implemented automatically, it is possible to systematically produce and evaluate many more options than could reasonably be done by hand. In doing so, we are able to demonstrate for every input size a range of designs with different cost and performance tradeoffs.

Related work. Most previous work on hardware implementations of non-power-of-two sized DFTs has focused on producing a solution for a specific situation (a given problem size and performance requirement). For example, [5] and [6] consider the specific problem sizes and performance requirements of the Digital Radio Mondiale (DRM) radio broadcasting standard, while [7] presents an FPGA-based pipeline design of a 3,780 point inverse DFT used in a digital television standard.

2. BACKGROUND

In this section, we present relevant background on the discrete Fourier transform and fast Fourier transform algorithms.

Discrete Fourier transform. Computing the discrete Fourier transform on n points is defined as the matrix-vector product $y = \mathrm{DFT}_n\, x$, where x and y are n point input and output vectors (respectively), and

$$\mathrm{DFT}_n = [\omega_n^{k\ell}]_{0\le k,\ell<n}, \quad \omega_n = e^{-2\pi i/n}.$$

Fast Fourier transform. Fast Fourier transform (FFT) algorithms allow the computation of DFT_n in $O(n \log n)$ operations. Using the Kronecker product formalism in [4], an FFT algorithm can be written as a *formula* that represents a factorization of the dense DFT_n matrix into a product of structured sparse matrices. For example, the well-known Cooley-Tukey FFT can be written as

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) T_n^{mn} (I_m \otimes \mathrm{DFT}_n) L_m^{mn}. \qquad (1)$$

Here, L_m^{mn} represents a stride permutation matrix that permutes its input according to:

$$in + j \mapsto jm + i, \quad 0 \le i < m,\ 0 \le j < n.$$

This work was supported by DARPA under the DOI grant NBCH1050009 and the ARO grant W911NF0710416 and by NSF awards 0325687 and 0702386.

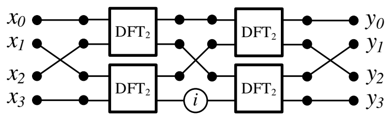


Fig. 1. Dataflow illustration of (1).

Further, I_m is the $m \times m$ identity matrix, and $\otimes$ is the *tensor product*, defined as

$$A \otimes B = [a_{k,\ell}B], \text{ where } A = [a_{k,\ell}].$$

Lastly, T_n^{mn} is a diagonal matrix of "twiddle factors," (defined, e.g., in [8]). Figure 1 shows a dataflow interpretation of the formula (1) when $n = 2$ and $m = 2$.

Automatic Hardware Generation. Spiral [1, 2] is an automated tool for the generation and optimization of hardware and software implementations of linear transforms including the DFT. Spiral takes a given transform matrix (DFT_n) and expands it recursively using one or more FFTs, resulting in a *formula*, which is subsequently optimized and then translated into code. Search enables the selection of the best solution. In [1], we extended the Kronecker formalism using a tagging framework to enable the description of a rich space of hardware architectures by explicitly specifying *sequential reuse* of computational elements within the formula. This allows Spiral to automatically generate many possible hardware implementations of a transform of a given size, each with different trade-offs between various costs (e.g., circuit area) and performance metrics (e.g., throughput, latency).

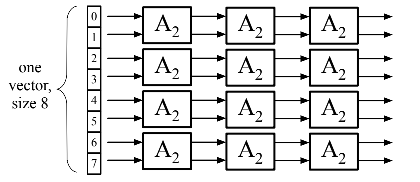
Figure 2(a) shows a simplified datapath with no sequential reuse. It contains three repeated stages, each with four parallel computation blocks labeled A_2. It processes eight data elements in parallel. Next, Figure 2(b) employs *streaming reuse* to stream the data vector through the system over multiple cycles at a fixed rate, called the *streaming width* (two in this example).[1] In Figure 2(c), the three computation stages have been collapsed to a single stage using a technique called *iterative reuse*. Now, the data vector must feed back and pass through the computation block multiple times.

By using different amounts of sequential reuse, different cost/performance trade-offs can be obtained. The more reuse employed, the smaller a datapath will be, but it will be commensurately slower. In [1], we provide more information on our formula-driven hardware generation techniques and evaluate the generated designs.

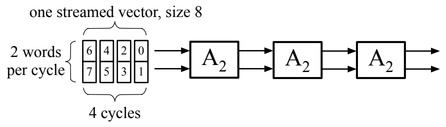
3. NON-POWER-OF-TWO FFT ALGORITHMS

In this section, we discuss four FFT algorithms that we use with our formula-driven generation framework to implement the DFT with non-power-of-two problem size.

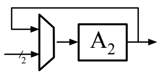
Pease FFT and Iterative FFT. The Pease FFT [11] and Iterative FFT are commonly used FFT algorithms that decompose an r^t point DFT into DFTs on r points, where r is called the *radix*. Both algorithms can be derived from (1). The radix-r Pease and Iterative FFTs are (respectively):



(a) Example combinational datapath.



(b) Datapath employing streaming reuse.



(c) Datapath employing iterative reuse.

Fig. 2. Illustration of sequential reuse techniques.

$$\mathrm{DFT}_{r^t} = \left(\prod_{\ell=0}^{t-1} L_r^{r^t} (I_{r^{t-1}} \otimes \mathrm{DFT}_r) D_\ell^{r^t} \right) R_r^{r^t} \tag{2}$$

$$\mathrm{DFT}_{r^t} = \left(\prod_{\ell=0}^{t-1} (I_{r^\ell} \otimes \mathrm{DFT}_r \otimes I_{r^{t-\ell-1}}) E_\ell^{r^t} \right) R_r^{r^t} \tag{3}$$

The matrix $R_r^{r^t}$ is the base-r digit reversal permutation on r^t points. When r = 2, this is called the "bit reversal permutation." D and E are diagonal matrices of twiddle factors. The remaining matrices are as defined in Section 2.

We apply streaming reuse to both algorithms (as in Figure 2(b)), but we only use (2) for iterative reuse (as in Figure 2(c)). When using these algorithms to generate streaming hardware, the streaming width must be a power of r. So, as r increases, the space of datapaths that can be generated by these formulas decreases.

Mixed radix FFT. The mixed radix FFT algorithm is a variant of (1); it decomposes a DFT of size $r^k s^\ell$ into multiple DFTs of sizes r^k and s^ℓ:

$$\mathrm{DFT}_{r^k s^\ell} = L_{r^k}^{r^k s^\ell} (I_{s^\ell} \otimes \mathrm{DFT}_{r^k}) L_{s^\ell}^{r^k s^\ell} T_{s^\ell}^{r^k s^\ell} \cdot (I_{r^k} \otimes \mathrm{DFT}_{s^\ell}) L_{r^k}^{r^k s^\ell}. \tag{4}$$

The smaller DFT_{r^k} and DFT_{s^ℓ} matrices can then be recursively decomposed using the radix-r and -s algorithms (2) or (3). The two computation stages corresponding to the two tensor products in (4) must be built independently, i.e., they can not be collapsed into one iterative reuse stage as in Figure 2(c).

The structure of (3) imposes an additional restriction on the applicable streaming width: it must evenly divide the

[1]Permutations on streaming data require the use of memory; we implement them as explained in [9, 10].

problem size $r^k s^\ell$, and it must be a multiple of rs. Thus, as the radices r and s get larger, the datapath options become more restricted.

Bluestein FFT. The Bluestein FFT [3] is a convolution-based algorithm for any problem size n. For a given n, the algorithm reduces the DFT to a circular convolution of two vectors of length $m > 2n - 1$. This convolution in turn can then be performed using two DFT_ms. Typically m is chosen as the smallest admissible 2-power. We can represent the Bluestein FFT as

$$\mathrm{DFT}_n = D^{(2)}_{n\times m}\,\mathrm{DFT}^{-1}_m\,D^{(1)}_m\,\mathrm{DFT}_m\,D^{(0)}_{m\times n}, \qquad (5)$$

where $m = 2^{\lceil \log_2(n)\rceil+1}$ is the smallest power of two greater than $2n-1$, and the D matrices are diagonal matrices that scale the vector by constant values. $D^{(0)}_{m\times n}$ and $D^{(2)}_{n\times m}$ are non-square matrices, so in addition to scaling the data, $D^{(0)}$ extends the input data vector from n points to m points (by zero padding), and $D^{(2)}$ shortens the output data vector from m points to n points (by discarding unneeded data).

The Bluestein FFT algorithm has a higher operation count than the other algorithms considered, but its structure is more regular than (4); the forward and inverse DFT_n can be collapsed into one logic block based on iterative reuse as in Figure 2(c). In Section 4 we will see that this way the Bluestein algorithm can yield smaller (but slower) hardware implementations than the mixed radix FFT.

4. EXPERIMENTAL RESULTS

In this section, we evaluate the designs generated from (2), (3), (4), and (5). We compare the effects of each algorithm and illustrate how the algorithmic options affect the quality and types of datapaths that can be implemented within our framework. Our experiments evaluate the designs implemented on a Xilinx Virtex-5 FPGA (field-programmable gate array). However, the designs produced by our tool are not FPGA-specific but are also suitable for implementation as an ASIC (application-specific integrated circuit).

Experimental Setup. Given a transform of a specific size, a set of algorithmic options, and hardware directives that specify the desired datapath characteristics, Spiral automatically generates a corresponding pipelined hardware implementation in synthesizable register-transfer level Verilog. In these experiments, we generate designs that operate on 16 bit fixed point data words (for each of the real and imaginary parts of complex data), but Spiral is able to generate designs for any precision fixed-point or single precision floating point.

In these experiments, we generate and evaluate all of the options presented above with a streaming width up to 32. We use Xilinx ISE to synthesize and place/route each design for a Xilinx Virtex-5 LX 330 FPGA. When a memory structure is needed, we utilize an on-chip block RAM when we will utilize 2KB or more, otherwise we build the memory out of FPGA logic elements. (This threshold is also a controllable parameter in our generation tool.)

The set of algorithms and datapaths we consider depends on the problem size. Here, we consider four values of n that illustrate four different classes of problem size. In Figure 3, we examine the throughput performance (in million samples per second) versus FPGA area consumed (in Virtex-5 slices) for each design. Data labels are used to indicate the number of Virtex-5 DSP slices consumed by each point in the Pareto-optimal set. A similar tradeoff evaluation can be performed for other cost or performance metrics, such as latency.

Large prime size. First, Figure 3(a) shows throughput versus area for implementations of DFT_{499}. Because 499 is a large prime number, the only applicable option among those we consider is the Bluestein FFT algorithm. Each black circle represents one Bluestein-based datapath, and the black line illustrates the designs in the *Pareto-optimal* set: the set of best tradeoff points among those evaluated. The slowest design requires approximately 1,500 slices and has a throughput of 11 million samples per second (MSPS). The fastest is about $18\times$ larger but $60\times$ faster.

Composite number with larger prime factors. Next, Figure 3(b) shows results for DFT_{405}. In addition to the Bluestein algorithm, we apply the mixed radix FFT algorithm to this problem; it decomposes DFT_{405} into DFT_{3^4} and DFT_5. This enables two additional designs (with streaming width $3 \cdot 5 = 15$), shown as white triangles. Although the larger of these designs provides a much higher throughput than a similarly sized Bluestein-derived design, it is important to note that the cross-over point (where the mixed-radix designs become the better choice) is quite large (at approximately 15,000 slices). So, if design requirements require a smaller, lower throughput core, the Bluestein algorithm is a better choice.

Composite number with smaller prime factors. Third, Figure 3(c) illustrates throughput and area for DFT_{432}. Similar to the previous example, DFT_{432} can be implemented using the mixed radix algorithm. However, unlike DFT_{405}, we can decompose DFT_{432} into smaller radices: 2 and 3 (because $432 = 2^4 \cdot 3^3$). Since the radices are smaller, the mixed radix algorithm can be used with more options for streaming width, yielding a larger set of Pareto-optimal designs than for DFT_{404}. The Bluestein designs are still the best choice for the smallest/slowest designs, but the cross-over point (about 9,000 slices) is lower than in the previous example.

Power of small prime. Lastly, Figure 3(d) shows results for $\mathrm{DFT}_{243} = \mathrm{DFT}_{3^5}$. Because the problem size is a power of three, the radix-3 Pease and Iterative FFTs are applied directly (shown as white triangles). Here, the cores built with the radix-3 algorithms surpass the performance of the Bluestein cores at a much lower area than in the other problems, since the single-radix design can be built with less logic than in the previous multi-radix designs.

These results show that our framework produces designs

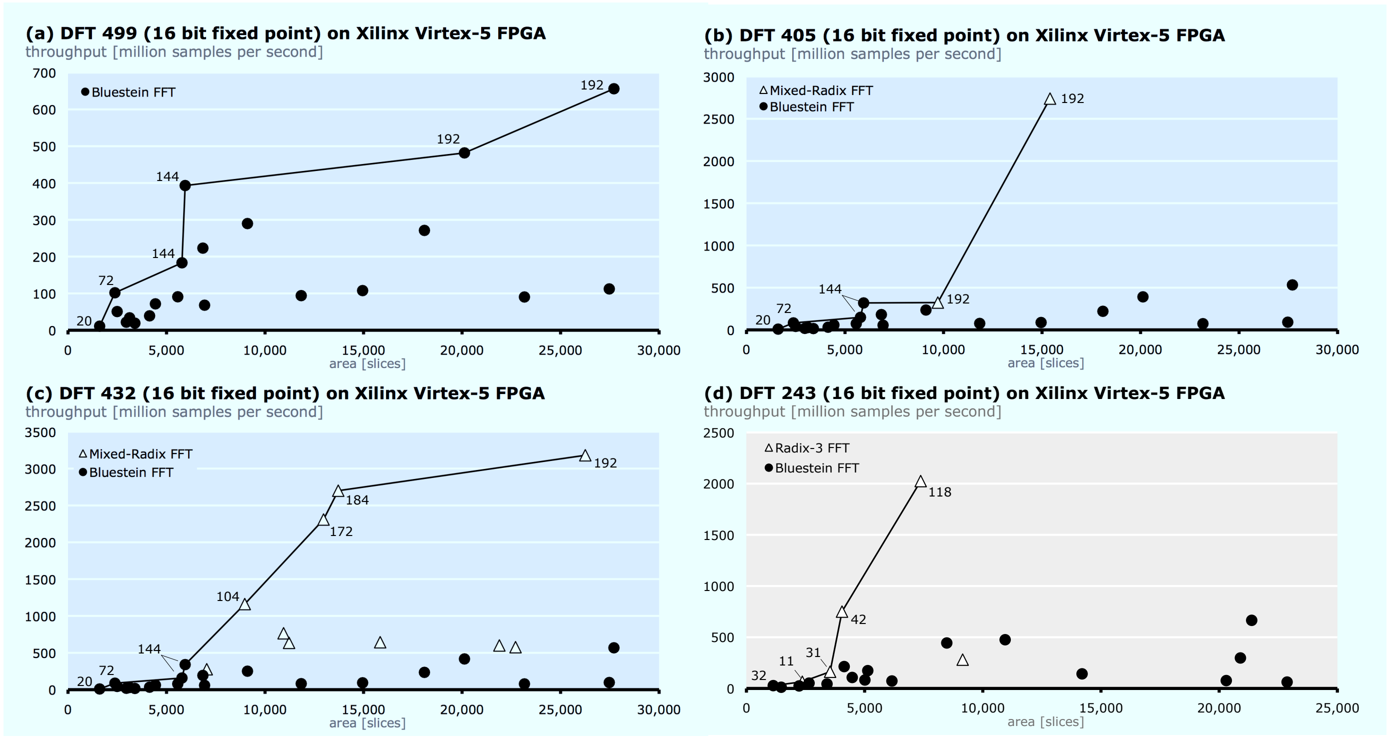


Fig. 3. Throughput (y-axis) versus area (x-axis) for DFT 499, 405, 432, and 243. Data labels: number of DSP slices.

with a range of cost/performance trade-offs. This range depends on the number-theoretic properties of the problem size, which affects the set of algorithms and streaming options that are applicable.

5. CONCLUSION

In this paper, we explore hardware implementation of the discrete Fourier transform with problem sizes that are not powers of two. We discuss several algorithms for this class of problems that fit within our automatic hardware generation framework Spiral. We show how different algorithms can be applied to DFTs of different problem sizes, and we discuss the types of hardware structures that may be generated for each. Lastly, we provide an evaluation of designs generated using the algorithmic and datapath options described, and show that the options considered allow the tool to produce designs over a range of cost/performance tradeoff points.

6. REFERENCES

[1] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Formal datapath representation and manipulation for implementing DSP transforms," in *Proc. Design Automation Conference*, 2008, pp. 385–390.

[2] M. Püschel et al., "SPIRAL: Code generation for DSP transforms," *Proc. of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005.

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

[4] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, SIAM, 1992.

[5] M. D. van de Burgwal, P. T. Wolkotte, and G. J. M. Smit, "Non-power-of-two FFTs: Exploring the flexibility of the Montium TP," *International Journal of Reconfigurable Computing*, vol. 2009, 2009.

[6] D.-S. Kim, S.-S. Lee, J.-Y. Song, K.-Y. Wang, and D.-J. Chung, "Design of a mixed prime factor FFT for portable digital radio mondiale receiver," *IEEE Trans. on Consumer Electronics*, vol. 54, no. 4, pp. 1590–1594, 2008.

[7] Z.-X. Yang, Y.-P. Hu, C.-Y. Pan, and L. Yang, "Design of a 3780-point IFFT processor for TDS-OFDM," *IEEE Trans. on Broadcasting*, vol. 48, no. 1, pp. 57–61, 2002.

[8] J. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures," *IEEE Trans. Circuits and Systems*, vol. 9, pp. 449–500, 1990.

[9] M. Püschel, P. A. Milder, and J. C. Hoe, "Permuting streaming data using RAMs," *Journal of the ACM*, vol. 56, no. 2, pp. 10:1–10:34, 2009.

[10] P. A. Milder, J. C. Hoe, and M. Püschel, "Automatic generation of streaming datapaths for arbitrary fixed permutations," in *Proc. Design, Automation and Test in Europe*, 2009, pp. 1118–1123.

[11] M. C. Pease, "An adaptation of the fast Fourier transform for parallel processing," *Journal of the ACM*, vol. 15, no. 2, April 1968.