Discrete Fourier Transform Compiler: From Mathematical Representation to Efficient Hardware

Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA, U.S.A.
{pam, franzf, jhoe, pueschel}@ece.cmu.edu

***Abstract*— A wide range of hardware implementations are possible for the discrete Fourier transform (DFT), offering different tradeoffs in throughput, latency and cost. The well-understood structure of DFT algorithms makes possible a fully automatic synthesis framework that can span the viable interesting design choices. In this paper, we present such a synthesis framework that starts from formal mathematical formulas of a general class of fast DFT algorithms and produces performance and cost efficient sequential hardware implementations, making design decisions and tradeoffs according to user specified high-level preferences. We present evaluations to demonstrate the variety of supported implementations and the cost/performance tradeoffs they allow.**

I. Introduction

The discrete Fourier transform (DFT) is one of the ubiquitous building blocks in signal processing and other embedded processing applications. Its computation exhibits a high degree of regularity in structure, comprising recurring basic kernels. On one hand, the theories behind efficient hardware implementations have been studied extensively and are very well understood [1]. On the other, creating practical implementations remains challenging in practice because it requires combined sophistication in the mathematics of transforms as well as in digital design.

When a design calls for a discrete Fourier transform, designers most often resort to instantiating pre-designed library implementations. Ready-to-use DFT modules are in the repertoire of nearly every technology vendor library—whether ASIC or FPGAs. These library modules are designed by specialists and generally attain optimum performance for the amount of resources they consume. However optimized, these static library modules can fall far short of optimum in a given application context due to a mismatch in objectives. For example, a static library module would offer exactly the same level of performance regardless of how much surplus logic resources are available. To address this limitation, Nordin et al. [2] have made available a parameterized DFT module generator that allows control over the level of hardware parallelism such that the designer can make custom tradeoffs between the performance desired and the resources consumed in the generated module.

Given the well-understood regular structure of the DFT (and other linear DSP transforms in general), one should be able to fully capture the available design space in a synthesis system to fully automate the generation of high-quality hardware implementations. The parameterized generation engine in [2] is a step in the right direction. However, this technology is limited in that the engine is hard-coded for a specific DFT algorithm (Pease [3]) and only exploits one particular restructuring to derive the tradeoff in one specific dimension of the overall design space.

In this paper, we present a formula-to-hardware synthesis flow that accepts as input the mathematical representation of a general class of DFT transform algorithms and is capable of producing a wide range of correct hardware implementations (in synthesizable RTL Verilog), including latency-efficient iterative microarchitectures and throughput-efficient streaming microarchitectures. The input representation is based on a sparse-matrix formula language. Starting from pure mathematical formulas, the synthesis process—comprising a set of formal formula-level rewrite rules—makes hardware implementation decisions and tradeoffs according to user specified high-level preferences. The outcome is a set of fully annotated formulas that can be straightforwardly reduced to its corresponding hardware sequential implementations. The wealth of initial DFT formula choices and the rich combinations of structural rewrite rules together yield the large space of implementations attainable by this DFT synthesis framework.

Paper outline. Section II introduces the DFT transform and the formula language and sketches a rudimentary synthesis algorithm for combinational implementations. Section III presents the flow from formula to hardware synthesis in two parts: first, from formula to annotated hardware formula, second, from hardware formula to sequential datapath. Section IV reviews the synthesis flow by demonstrating two working examples. Section V evaluates a wide range of DFT design instances produced by our framework. Section VI discusses prior work in hardware DFT implementations. Finally, Section VII offers our discussion and conclusions.

II. Background

Discrete Fourier transform. The discrete Fourier transform (DFT) of size n is the matrix-vector multiplication

$$y = \mathrm{DFT}_n\, x,$$

where x and y are the input and output vectors of length n, and

$$\mathrm{DFT}_n = [\omega_n^{k\ell}]_{0\le k,\ell<n}, \quad \omega_n = \exp(-2\pi i/n),\ i = \sqrt{-1}.$$

In this paper we only consider two-power sizes n.

Fast Fourier transforms. Computing the DFT of x by matrix-vector multiply requires $\mathrm{O}(n^2)$ operations. However, this can be reduced to $\mathrm{O}(n\log(n))$ using well-known fast algorithms (fast Fourier transforms or FFTs). An FFT can be viewed as a factorization of DFT_n into a product of sparse matrices. For example (omitted entries are zero):

$$\mathrm{DFT}_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} = \begin{bmatrix} 1 & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & 1 \\ 1 & \cdot & -1 & \cdot \\ \cdot & 1 & \cdot & -1 \end{bmatrix}$$
$$\cdot \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & \cdot & \cdot & i \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdot & \cdot \\ 1 & -1 & \cdot & \cdot \\ \cdot & \cdot & 1 & 1 \\ \cdot & \cdot & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 1 & \cdot \\ \cdot & 1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 1 \end{bmatrix}$$

Computing $\mathrm{DFT}_4\, x$ by multiplying the input x from right to left with the four sparse matrices has a lower arithmetic cost than multiplying by the dense transform matrix.

Each of the occurring sparse matrices has structure, which can be used to express the algorithm using the Kronecker or tensor product formalism [1]. For example, the factorization above becomes

$$\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes I_2) D_4 (I_2 \otimes \mathrm{DFT}_2) L_{4,2}. \tag{1}$$

Here $D_4 = \mathrm{diag}(1,1,1,i)$ is a diagonal matrix (called *twiddle matrix*); I_n the $n\times n$ identity; $L_{n,m}$ (for m divides n) the stride permutation, which can be viewed as transposing an $n/m \times m$ matrix stored in a vector in row-major order, formally:

$$i\cdot(n/m) + j \mapsto j\cdot m + i, \quad \text{for } 0\le i<m,\ 0\le j<n/m.$$

Finally, $\otimes$ is the Kronecker or tensor product defined as

$$A_n \otimes B_m = [a_{k,\ell} B_m]_{0\le k,\ell<n}, \quad \text{for } A = [a_{k,\ell}]_{0\le k,\ell<n}.$$

Generally, we write A_n to denote that A is $n \times n$.

Formula language for FFTs. The above formalism can be captured in a formal language that can be used to represent FFTs using *formulas*. In Backus-Naur form, the language is defined as follows (non-terminals are bold-faced):

$$\begin{aligned} \mathbf{formula}_n ::= & \mathbf{formula}_n \cdots \mathbf{formula}_n \\ & \mid I_k \otimes \mathbf{formula}_m && \text{where } n = km \\ & \mid \mathbf{formula}_m \otimes I_k && \text{where } n = km \\ & \mid \mathbf{base}_n \end{aligned}$$

$$\mathbf{base}_n ::= D_n = \mathrm{diag}(d_0,\ldots,d_{n-1}) \mid L_{n,m} \mid I_n \mid \mathrm{DFT}_2$$

This language is a subset of the signal processing language (SPL) used in Spiral, a program generator for linear transforms [4]. We will also refer to it as SPL in this paper.

Even though the language is small, a large class of different FFTs can be expressed with it. We provide a few examples:

$$\mathrm{DFT}_{nm} = (\mathrm{DFT}_n \otimes I_m) D_{n,m} (I_n \otimes \mathrm{DFT}_m) L_{nm,n} \tag{2}$$

$$\begin{aligned} \mathrm{DFT}_{r^t} = {} & L_{r^t,r} \Big(\prod_{k=0}^{t-2} (I_{r^{t-1}} \otimes \mathrm{DFT}_r)\, D_{n,k} \\ & (I_{r^k} \otimes L_{r^{t-k},r^{t-k-1}})(I_{r^{k+1}} \otimes L_{r^{t-k-1},r}) \Big) \\ & (I_{r^{t-1}} \otimes \mathrm{DFT}_r)\, R_{r^t,r} \end{aligned} \tag{3}$$

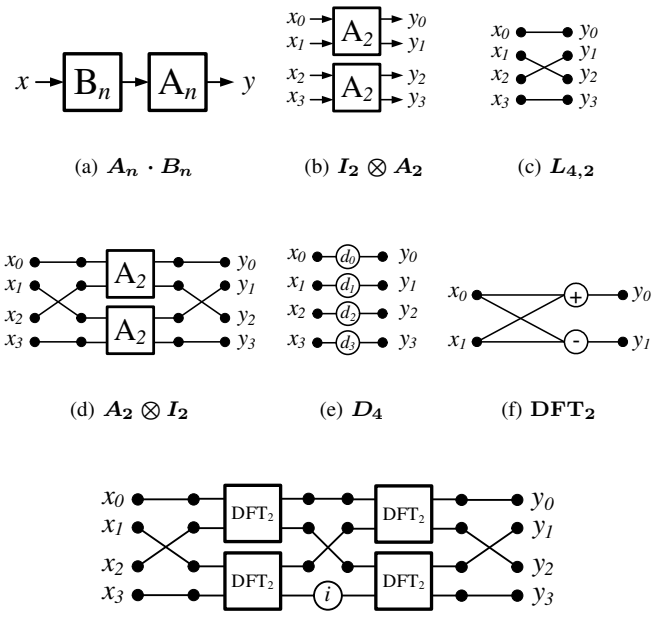
$$\mathrm{DFT}_{r^t} = \left[\prod_{k=0}^{t-1} L_{r^t,r}\, (I_{r^{t-1}} \otimes \mathrm{DFT}_r)\, D_{n,k} \right] R_r^{r^t} \tag{4}$$

Equation (2) is the well-known recursive Cooley-Tukey FFT and the standard choice for software implementations. The others are iterative and suitable for hardware implementations. Equation (3) is called the iterative FFT, and (4) is the Pease FFT, which has perfect regularity across stages except for the diagonal matrices $D_{n,k}$ which depend on k. Both FFTs are given for an arbitrary radix r, where the radix indicates the size of the algorithm's basic block. Lastly, $R_{r^t,r}$ is the radix-r reversal permutation, known as the bit reversal when $r = 2$.

Formula to combinational datapath. There exists a natural one-to-one correspondence between an SPL formula and a combinational logic implementation. We demonstrate it for different formula constructs M_n supported by the grammar:

- $\boldsymbol{M_n = A_n \cdot B_n}$: The input vector x first passes through a combinational module corresponding to B then another module corresponding to A (Fig. 1(a)).
- $\boldsymbol{M_{mn} = I_m \otimes A_n}$: The resulting matrix is a block diagonal matrix that contains zero everywhere except A_n repeated m times on the diagonal. In combinational logic, the vector x passes through m parallel copies of A such that each copy operates on a flit of n consecutive elements from x (Fig. 1(b)).
- $\boldsymbol{M_{mn} = A_n \otimes I_m}$: We can first rewrite this matrix as $L_{nm,n}(I_m \otimes A_n)L_{mn,m}$ (a known identity [1]) and handle it as the product of three matrices (Fig. 1(d)).
- $\boldsymbol{M_n = L_{n,m}}$: The corresponding reordering of the vector x is achieved by reshuffling the busses that carry the elements of x (Fig. 1(c)).
- $\boldsymbol{M_n}$ **is diagonal**: Each element of x is multiplied by a corresponding constant (Fig. 1(e)).
- $\boldsymbol{M_n = \mathrm{DFT}_2}$: This incurs the computations $y_0 = x_0 + x_1$ and $y_1 = x_0 - x_1$ and yields the so-called "butterfly" structure in Fig. 1(f).

Fig. 1 shows that it would be a straightforward task to generate combinational logic for any formula in the SPL language. For example, for the formula in (1), this compiler would produce the datapath shown in Fig. 1(g). For general n, the FFTs (3) and (4) would yield combinational logic that is $\mathrm{O}(\log(n))$ in depth and $\mathrm{O}(n\log(n))$ in size. Such combinational implementations are too expensive except for small problem sizes.



(a) $\boldsymbol{A_n \cdot B_n}$ (b) $\boldsymbol{I_2 \otimes A_2}$ (c) $\boldsymbol{L_{4,2}}$

(d) $\boldsymbol{A_2 \otimes I_2}$ (e) $\boldsymbol{D_4}$ (f) $\mathbf{DFT_2}$

(g) $\mathbf{DFT_4}$ (from (1))

Fig. 1. Examples of formulas and associated combinational datapaths.

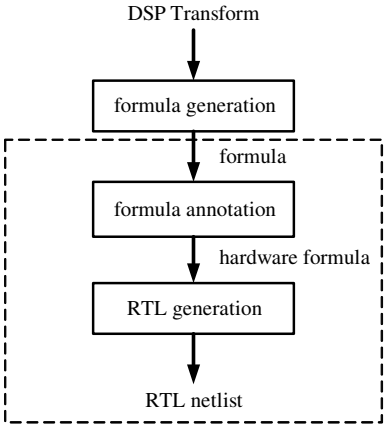
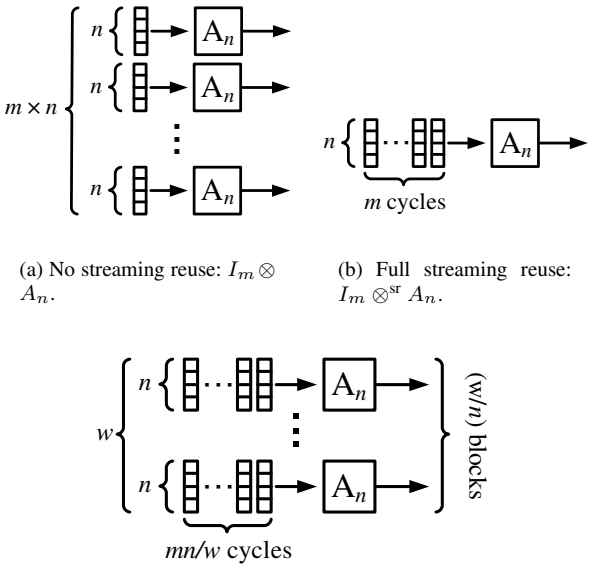


Fig. 2. Block diagram of design flow. The dashed block contains the focus of this paper.

III. From Formula to Hardware

Our goal is to automatically generate various sequential implementations of the DFT. Our formula language, as explained in Section II, has no singular correspondence to sequential hardware. In this section, we explain how we extend this language to express sequential hardware elements needed for efficient implementations. Then, we introduce a rewriting system which takes a formula with *hardware directives* and produces a hardware description formula. Lastly, we discuss the process of compiling a hardware description formula to a synthesizable RTL netlist. This flow is illustrated by the diagram in Figure 2.



(a) No streaming reuse: $I_m \otimes A_n$. (b) Full streaming reuse: $I_m \otimes^{\text{sr}} A_n$.

(c) Partial streaming reuse: $I_{mn/w} \otimes^{\text{sr}} (I_{w/n} \otimes A_n)$.

Fig. 3. Examples of streaming reuse.

A. Hardware Signal Processing Language

The datapaths associated with various DFT algorithms exhibit a high degree of regularity. This regular structure gives an opportunity to reuse portions of the datapath in two ways. In this section, we examine both types of reuse and the language extensions needed to support them.

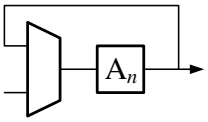
Streaming reuse. As seen in Section II, the tensor product $I_m \otimes A_n$ leads to a datapath with m data-parallel instances of the block associated with A_n (Fig. 3(a)). We also can interpret the tensor product as an indicator of parallelism *in time* in a streaming fashion. Rather than having block A_n repeated m times in parallel, we can build one physical instance of it, and reuse it over m consecutive clock cycles (Figure 3(b)). We call this *streaming reuse*. In order to distinguish between these two meanings of the tensor product, we can tag the symbol $\otimes^{\text{sr}}$ in order to indicate streaming reuse.

Additionally, it is possible to have partial streaming reuse. For example, $(I_4 \otimes A_4)$ can be broken down as $(I_2 \otimes^{\text{sr}} (I_2 \otimes A_4))$, meaning that there are two A_n blocks in parallel, and each operates on data over two clock cycles. A generalized version of this situation can be seen in Figure 3(c). We use w to indicate the stream width.

Horizontal reuse. In the previous section, we saw how data parallel blocks could be vertically collapsed into one block. Additionally, a series of identical blocks (such as $\prod A_n$) can be horizontally collapsed into one block, as seen in Figure 4. We call this *horizontal reuse*. (Notice that A_n could be streamed as well.) In order to distinguish between the two meanings of $\prod$, we tag the product term $\prod^{\text{hr}}$ in order to indicate horizontal reuse.

It is important to note that the terms in a horizontal reuse

(a) No horizontal reuse: $\prod_4 A_n$.



(b) Full horizontal reuse: $\prod_4^{\text{hr}} A_n$.

Fig. 4. Example of horizontal reuse.

product term cannot change from iteration to iteration, except in the case of diagonal matrices. For example, (3) would not be eligible, but (4) would. This is explained in detail in Section III-C.

B. Rewriting System: From Transform to Hardware Formula

In this section, we describe a rewriting system that takes a formula plus *hardware directives* and produces a hardware formula, which is restructured and annotated such that it directly corresponds to a sequential hardware implementation. This process corresponds to the "formula annotation" segment of Figure 2.

A hardware directive is a tag that indicates a desired feature of the final hardware implementation. In order to indicate streaming reuse, we define a streaming tag:

$$\underbrace{A_n}_{\text{stream}(w)}$$

This tag indicates that the contents of A should be restructured such that the resulting hardware formula will be implemented in a block that contains w input and output ports, with data streamed at w elements per cycle. Figure 5 lists the rewriting rules that perform this transformation. Each time the system encounters a stream tagged formula, it attempts to restructure the formula or propagate the tag downward. If a tagged formula does not match any of these rules, the tag becomes part of the hardware formula. In these cases, the compiler (discussed in the following section) must explicitly know how to build a data structure for the tagged formula.

Each of the rewrite rules given in Figure 5 has a simple explanation:

- **base:** If the size of a matrix is the same as its stream size, the stream tag is not necessary and can be dropped.
- **product select:** This rule selects whether to do horizontal reuse or streaming.
- **product and product HR:** If a group of matrices is tagged as streaming, the tag is propagated inward to all of the individual matrices. This rule applies to both versions of the product term $\prod$.

name	rule
base	if $k = w$, $\underbrace{A_k}_{\text{stream}(w)} \rightarrow A_k$
product-select	$\underbrace{\prod A_n}_{\text{stream}(w)} \rightarrow \begin{cases} \underbrace{A_0 \cdot A_1 \cdots A_n}_{\text{stream}(w)} & \text{if streaming} \\ \underbrace{\prod^{\text{hr}} A_n}_{\text{stream}(w)} & \text{if hor. reuse} \end{cases}$
product	$\underbrace{A_n \cdot B_n \cdots Z_n}_{\text{stream}(w)} \rightarrow \underbrace{A_n}_{\text{stream}(w)} \cdot \underbrace{B_n}_{\text{stream}(w)} \cdots \underbrace{Z_n}_{\text{stream}(w)}$
product-HR	$\underbrace{\prod^{\text{hr}} A_n}_{\text{stream}(w)} \rightarrow \prod^{\text{hr}} \underbrace{A_n}_{\text{stream}(w)}$
reuse1	if $\ell k > w$ and $k \leq w$, $\underbrace{I_\ell \otimes A_k}_{\text{stream}(w)} \rightarrow I_{\ell/w} \otimes^{\text{sr}} \left(I_{w/k} \otimes A_k\right)$
reuse2	if $k > w$, $\underbrace{I_\ell \otimes A_k}_{\text{stream}(w)} \rightarrow I_\ell \otimes^{\text{sr}} \underbrace{A_k}_{\text{stream}(w)}$
reverse	$\underbrace{A_k \otimes I_\ell}_{\text{stream}(w)} \rightarrow \underbrace{L_{k\ell,k} \left(I_\ell \otimes A_k\right) L_{k\ell,\ell}}_{\text{stream}(w)}$

Fig. 5. Rewriting rules for generating hardware formulas.

- **reuse1:** If the size of A is less than or equal to the size of the stream, the inner tensor product unrolls the correct number of A instances such that the inner product is exactly the stream size.
- **reuse2:** If the size of A is larger than the size of the stream, the tag is propagated inward, and another rule must restructure A to the right stream size.
- **reverse:** A property of the tensor product allows us to reverse its order with strided access.

After rewriting, the resulting hardware formula may be made of the following blocks: formulas (without tags), streaming-reuse tensor products, horizontal-reuse product terms, streamed diagonals, and streamed permutations (i.e., stride and bit reversal permutations):

$$A, \quad I_\ell \otimes^{\text{sr}} A, \quad \prod^{\text{hr}} A_n, \quad \underbrace{D_n}_{\text{stream}(w)}, \quad \underbrace{L_{n,m}}_{\text{stream}(w)}, \quad \underbrace{R_r^n}_{\text{stream}(w)}.$$

In the following section, we will discuss how the hardware formula, made of these five types of objects, is built into a Verilog description.

C. Compiler: From Hardware Formula to HDL

The compiler takes in a hardware formula, as defined above, and produces a synthesizable Verilog description of a circuit. In this section, we explain how each of the possible forms of the hardware formula is mapped.

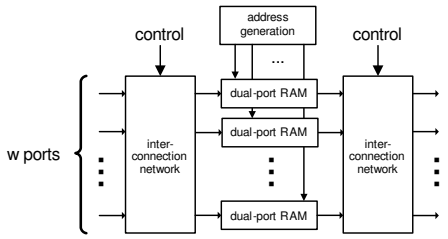


Fig. 6. Structure for permuting streamed vector elements.

Combinational formula. Any portion of the formula without a reuse tag can automatically be mapped into a combinational datapath, as discussed in Section II. When the compiler encounters this type of formula, it constructs a hardware datapath and automatically pipelines the path by inserting staging registers in the appropriate locations.

Specific streaming elements. We implement two elements that are built directly from a stream-tagged matrix. The compiler has specific knowledge of how to generate these blocks:

- **Streaming diagonal:** A diagonal matrix scales each element in the input vector by the corresponding value from the diagonal of the matrix. In order to convert this into a streaming hardware structure, we first need w multipliers, where w is the stream width. Then, we store the n values from the diagonal in w lookup tables, which feed the multipliers with the appropriate data at each cycle.
- **Streaming permutations:** A streaming permutation implementation must reorder data in space and across different clock cycles. Püschel et al. [5] prescribe an architecture and an algorithm whereby an arbitrary permutation can be constructed for an arbitrary streaming width w (where both the vector length n and stream width w are 2-powers). The construction, sketched in Figure 6, uses w dual-ported memory banks and configurable switching networks at the input and output stages. For the relevant permutations, the controls for the permutation block can be computed cheaply from the flit number using only bitwise operators. For example, Figure 7 shows an implementation of $L_{256,2}$ with streaming width $w = 4$. Because this method works with a general class of permutations, it is able to implement products of permutations (e.g., $L_{8,4} \cdot (I_2 \otimes L_{4,2})$) as one self-contained permutation module.

Finally, the compiler maps formulas tagged for horizontal or streaming reuse to the appropriate structural hardware construct:

- **Streaming reuse:** A streaming reuse tensor product $I_m \otimes^{\text{sr}} A_n$ is implemented in hardware as one A_n block intended for an input vector in a streamed format (as seen in Fig. 3(b).
- **Horizontal reuse:** As shown in Fig. 4(b), an horizontal reuse structure is built with an input multiplexer and feedback loop. When the block contains D_k, a diagonal matrix that changes with the iteration, the lookup table must grow to accommodate all values. If a data vector iterates ℓ times over this datapath, the diagonal table must grow by a factor of ℓ.

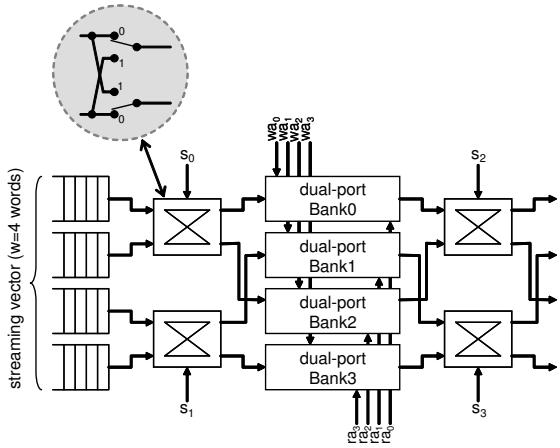


Fig. 7. Example of RAM-based permutation: $\boldsymbol{L_{256,2}}$ with 4 ports.

IV. Examples: From Formula to Datapath

We demonstrate the automated synthesis flow for two different DFT formulas.

Streaming reuse. The iterative FFT, given in Equation (3), produces a streaming reuse structure. For size 8 and radix 2, this formula[1] simplifies to:

$$\begin{aligned}\mathrm{DFT}'_{2^3} = L_{8,2}(I_4 \otimes \mathrm{DFT}_2)D_{8,0}L_{8,4}(I_2 \otimes L_{4,2}) \\ \cdot(I_4 \otimes \mathrm{DFT}_2)D_{8,1}(I_2 \otimes L_{4,2})(I_4 \otimes \mathrm{DFT}_2).\end{aligned}$$

The entire formula is then tagged as streaming with a stream size:

$$\begin{aligned}\underbrace{\mathrm{DFT}'_{2^3}}_{\text{stream(2)}} = \underbrace{L_{8,2}}_{\text{stream(2)}}\ \underbrace{(I_4 \otimes \mathrm{DFT}_2)}_{\text{stream(2)}}\ \underbrace{D_{8,0}}_{\text{stream(2)}}\ \underbrace{L_{8,4}(I_2 \otimes L_{4,2})}_{\text{stream(2)}} \\ \cdot \underbrace{(I_4 \otimes \mathrm{DFT}_2)}_{\text{stream(2)}}\ \underbrace{D_{8,1}}_{\text{stream(2)}}\ \underbrace{(I_2 \otimes L_{4,2})}_{\text{stream(2)}}\ \underbrace{(I_4 \otimes \mathrm{DFT}_2)}_{\text{stream(2)}}.\end{aligned}$$

Then, the rewrite system described in Section III-B changes the formula to a hardware description formula. For this example, the rewrite rules produce the following hardware formula:

$$\begin{aligned}\underbrace{\mathrm{DFT}'_{2^3}}_{\text{stream(2)}} = \underbrace{L_{8,2}}_{\text{stream(2)}}\ (I_4 \otimes^{\text{sr}} \mathrm{DFT}_2)\ \underbrace{D_{8,0}}_{\text{stream(2)}}\ \underbrace{L_{8,4}(I_2 \otimes L_{4,2})}_{\text{stream(2)}} \\ (I_4 \otimes^{\text{sr}} \mathrm{DFT}_2)(I_2 \otimes^{\text{sr}} \underbrace{L_{4,2}}_{\text{stream(2)}})\ \underbrace{D_{8,1}}_{\text{stream(2)}}\ (I_4 \otimes^{\text{sr}} \mathrm{DFT}_2). \quad (5)\end{aligned}$$

Each term in this equation is directly translated to a hardware datapath according to Section III-C (from the right of the formula to the left), producing the datapath seen in Fig. 8.

Streaming and horizontal reuse. The Pease FFT algorithm [3] given in Equation (4) produces an architecture with

[1]We implement the DFT with the digit reversal permutation R omitted. This is a common interface option in hardware DFT implementations. We indicate this in the formula as DFT'.

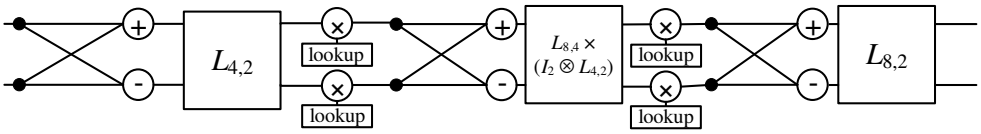


Fig. 8. Datapath implementation of streaming DFT_8 with stream size of 2

both horizontal and streaming reuse. For size 16 and radix 4, this formula simplifies to:

$$\mathrm{DFT}'_{4^2} = \prod_{k=0}^{1} L_{16,4}\,(I_4 \otimes \mathrm{DFT}_4)\,D_{16,k}$$

Next, this formula is tagged with a stream size. Additionally, the product can be tagged for horizontal reuse, because the formula inside it only contains iterator k in the diagonal matrix. The formula is then converted to a hardware formula, as in Section III-B.

If the stream size is set to 4, the following hardware formula is obtained:

$$\underbrace{\mathrm{DFT}'_{4^2}}_{\text{stream}(4)} = \prod_{k=0}^{1}{}^{\text{hr}}\ \underbrace{L_{16,4}}_{\text{stream}(4)}\,(I_4 \otimes^{\text{sr}} \mathrm{DFT}_4)\ \underbrace{D_{16,k}}_{\text{stream}(4)} \qquad (6)$$

Each term of this formula is translated directly to an RTL netlist (reading the formula from right to left), and the resulting datapath is shown in Fig. 9. However, this datapath includes two optimizations that are performed in the compiler:

The first optimization reduces the amount of arithmetic hardware that is built. Due to the structure of the diagonal matrix in the Pease algorithm, one out of every r multipliers will always access a value of 1. By labelling these values as trivial constants and giving the system some additional arithmetic simplification rules, the tool is able to determine that these multipliers will always multiply by 1 and thus remove them. This reduces the number of multipliers by 1 out of every r.

The second optimization allows the amount of lookup table data to be reduced by a factor of $\log_r(n)$. A horizontal reuse block with a diagonal matrix $D_{n,k}$ that changes with each iteration requires n elements to be stored for each of the $\log_r(n)$ iterations, leading to a storage requirement of $n \log_r(n)$ words. However, the diagonals in the Pease formula pose a special property: the set of all values of $D_{n,\ell}$ (the diagonal values at iteration ℓ) is a subset of the values of $D_{n,\ell-1}$ for $\ell > 0$. This means that with the right access function, all $n \log_r(n)$ data words can be obtained from a table of n words. When a stream tag is applied to the Pease diagonal matrix, our system recognizes this property. Then, it applies the correct access function to the representation and stores only the n data words corresponding to $D_{n,0}$. The new access function is very simple, consisting of bit-shifts and bitwise ANDs of indices. So, the Pease storage requirement is reduced by a factor of $\log_r(n)$.

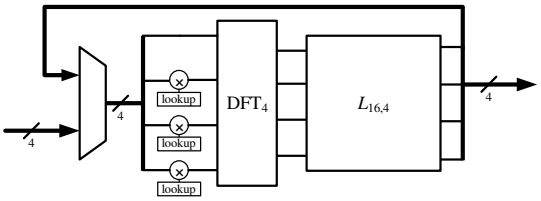


Fig. 9. Example of Pease DFT_{16} with $w = 4$.

V. Evaluation

When coupled with a formula generator like Spiral [4], the formula synthesis flow described in this paper enables a large number of DFT designs to be explored quickly in a turnkey fashion. This section evaluates the supported range of implementations and the different cost/performance tradeoffs they provide. As we offer designs over a wide range of tradeoffs between performance and cost, our evaluations include a comparison to the Xilinx LogiCore FFT implementation [6] to establish that the tradeoffs have a sound basis. Specifically, we select as reference three LogiCore FFT implementations: the radix-4 burst I/O implementation, the radix-2 minimal size implementation, and the pipelined streaming I/O implementation, each with a scaled fixed-point (16-bit) data format, and natural-in bit-reverse-out data ordering.

A. Methodology

We implemented the formula-to-hardware synthesis flow as a new hardware backend to Spiral's formula generator, which produced all the starting DFT formulas studied in this paper. The hardware-specific formula rewriting rules discussed in Section III-B are implemented as a part of Spiral's formula manipulation stage to produce the annotated hardware formulas. Finally, an RTL generator, implemented in Java, emits synthesizable Verilog RTL descriptions from the hardware formulas.

When an evaluation in this section reports synthesized results, the Verilog descriptions are synthesized and place-and-routed for the Xilinx XC2VP100-6 FPGA using Xilinx ISE 8.1. We report implementation cost in units of slices. All synthesized designs use 16-bit fix-point data format. To curtail synthesis load, we consistently use 7ns as the target

clock frequency.[2] Our RTL generator will use a block-RAM for storage if the usage will utilize more than 50 percent of that block-RAM; otherwise, distributed-RAM is used instead.[3]

B. Throughput Performance vs. Cost

We first evaluate the tradeoff between cost (in slices) and throughput performance (number of transforms completed per second). For DFT_{64} and DFT_{256}, we evaluate implementations of fully-streaming Iterative FFT and horizontal-reuse Pease FFT. For each algorithm/architecture combination, we explore radices[4] 2, 4, and 8. We include implementations with streaming width from $w = r$ up to the maximum allowed by the FPGA capacity. Their cost and throughput are reported in Figure 10. Throughput (y-axis) is presented in terms of *gap*, the time between starts in the steady-state. The x-axis indicates cost in slices. The plots show separate trend lines for each combination of algorithm, radix, and architecture. Each trend line begins (left to right) with streaming width ($w = r$) and doubles thereafter.

In the Pareto-style plot, points closer to the origin represent designs that are smaller and faster. Only points on the Pareto front—points that are not overshadowed by another point that is both faster and smaller—should be used in practice. It is important to note that the Pareto front comprises points arising from different combinations of algorithmic and architectural decisions.

In both DFT_{64} and DFT_{256}, the fully-streaming implementations based on Iterative FFT algorithm provide the fastest (yet commensurately more expensive) design points. For all radix choices, the results show an increase in throughput as more slices are consumed (by increasing streaming width w). Implementations using larger radices generally have better performance/cost ratios relative to comparable implementations based on lower radices. This is because, for the comparable choices of streaming width, all implementations consistently synthesize to comparable frequencies regardless of radix. Hence, all streaming implementations of DFT_{64} with the same stream width should achieve comparable throughput. On the other hand, for the same stream width, higher radix implementations have the advantage of fewer permutation and twiddle stages. However, the difference between radix 8 and 4 is much less noticeable than between radix 4 and 2.

The throughput evaluation includes horizontal-reuse implementations based on the Pease FFT to provide the very cheap but commensurately lower throughput design points. Gap is still measured in terms of time between starts of new DFT computations, but these horizontal-reuse implementations cannot support continuous streaming of vectors.

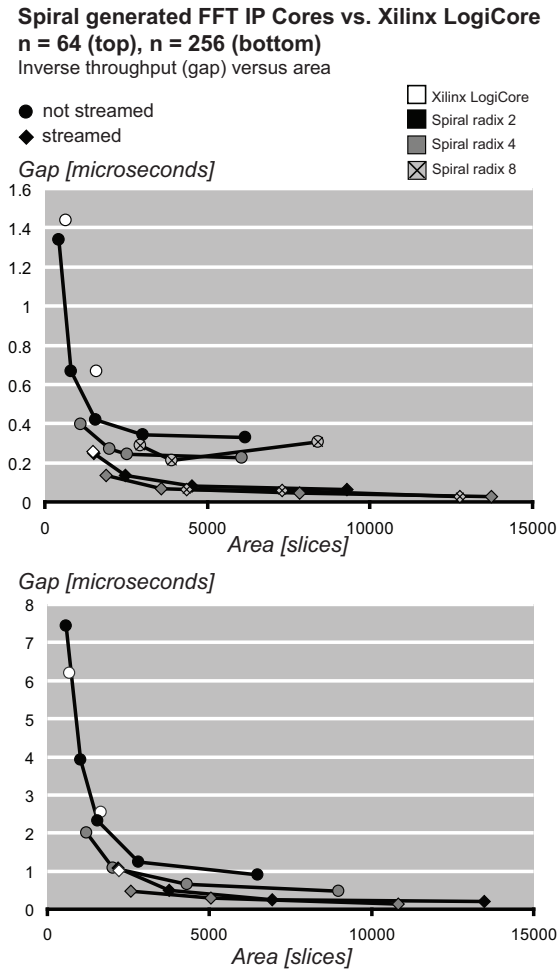


Fig. 10. Gap (1 / throughput) versus cost for implementations of $\mathbf{DFT_{64}}$ (top) and $\mathbf{DFT_{256}}$ (bottom).

Data points corresponding to the LogiCore FFT implementations are included in Figure 10. They serve as reference points to show that our designs are of good quality and yield a real increase in performance for the extra resources they consume.

C. Latency Performance vs. Cost

Next, we evaluate the tradeoff between cost (in slices) and latency performance (time elapsed for one transform computation). For DFT_{64}, DFT_{256}, and DFT_{1024}, we evaluate implementations of horizontal-reuse Pease FFT only. (Fully-streaming implementations are always Pareto sub-optimal in this regard because they are optimized for high throughput at the expense of extended latency.) We explore radices $r = \{2, 4, 8\}$ when $n = 64$; radices $r = \{2, 4\}$ for $n = 256$; and radices $r = \{2, 4\}$ for $n = 1024$. We include implementations from the minimum streaming width $w = r$ up to the maximum allowed by the FPGA capacity. The cost and latency are

[2]This is methodology is acceptable because for moderate streaming width $w \leq 8$, the cycle times of our DFT implementations are determined by critical paths in the complex arithmetic pipeline stages (consistently synthesize to between 7 and 8 ns). For the larger and wider designs, the synthesized frequency inherently becomes less predictable (typically 12 to 18 ns) due to routing and placement effects. Overall, our methodology makes our performance results conservative as our performance could possibly improve by choosing a different frequency target. When reporting synthesis results for the Xilinx LogiCore Library, we report the highest performing outcome from synthesizing their designs over a range of target frequencies.

[3]Block-RAM are the 16-kilobit memory hard macros in the Xilinx Virtex-II Pro FPGAs. Distributed-RAM consumes 16-bits per *slice*. In general, our generator lets the user set arbitrary switch-over point between using block-RAM vs distributed-RAM.

[4]The problem size n must be a power of r, the radix.

reported in Figure 11. For each design point, the y-axis indicates latency in microseconds, and the x-axis indicates cost in slices. Again in this Pareto-style plot, points closer to the origin are cheaper and faster. Similar to the previous reporting format, the plots show separate trend lines for each combination of algorithm/radix/architecture. Each trend line begins (left to right) with streaming width ($w = r$) and doubles thereafter.

For all radix choices, the horizontal-reuse implementations show a decrease in latency as more resources are consumed for wider stream width w. Again, a large improvement in performance/cost ratio is seen for radix-4 relative to radix-2 implementations[5], but the difference between radix-8 and radix-4 is less significant. Employing higher-radix implementations has another advantage that is more subtle. For example, to achieve the same latency, a radix-2 implementation needs approximately twice the streaming width as in a radix-4 implementation (to get the same amount of computation per cycle). These performance-comparable radix-2 and radix-4 implementations will also have comparable cost as well. (The same relationship exists between radix-4 and radix-8.) The subtle but important difference is that a $w = 4$ radix-4 implementation only requires loading and unloading 4 vector elements per cycle at the start and finish of each computation instead of 8 elements per cycle for a comparable performance radix-2 implementation. For the same cost and performance, a higher radix implementation is more desirable due to this lower interface bandwidth.

Again, data points corresponding to the LogiCore DFT implementations are shown in Figure 11 to provide a baseline. Our horizontal-reuse implementations allow more direct comparisons against LogiCore's latency and cost optimized architectures. Among our different latency/cost implementations, the low cost high latency implementations correspond most closely to LogiCore's tradeoff points.

D. Range of Implementations

Given the multi-variable and multi-objective nature of optimizing FFT implementations, it is impossible to completely explore the full range of designs or to properly compare tradeoffs across all combinations of metrics. In Table I, we highlight some of the most salient design corners attainable using the design choices described in this paper. Columns 1 to 5 specify the corresponding decisions used (problem size n, algorithm, radix r, architecture, stream width w). Columns 6 to 9 report the performance and cost metrics (throughput, latency, slices used, block-RAM used).

VI. Related Work

An extensive base of fundamental work in FFT algorithms and architectures for VLSI and FPGA has laid the foundation for this work. The mathematical framework described in this paper is capable of representing a wide variety of designs, incorporating optimizations at both the algorithmic and architectural levels.

Examples of prior work in fully-streamed (or pipelined) FFT implementations can be seen in [7], [8], and [9]. In some previous pipelined implementations, arithmetic units are not fully utilizable (e.g., [7] and [9]) due to their permutation implementations. Examples of prior work examining horizontal-reuse FFT implementations can be seen in [10] and [11]. Specifically, Pease FFTs with horizontal reuse are discussed in [2] and [12]. On the whole, many prior developments have covered much of the same design space we considered in this paper. However, these implementations were tuned for different objectives and targeted different technologies, preventing a systematic representation of the design space. Our study is somewhat unique in its extensive coverage of varied implementation parameters, using real RTL designs and real FPGA synthesis.

Below, we highlight examples of some important design choices not examined in this study. We did not consider the impact of fixed-point precision [13] or floating-point arithmetics [14]. We considered neither on-the-fly twiddle generation using CORDIC [15] nor distributed arithmetic to optimize the arithmetic pipeline at the bit-level [16]. We concentrated on performance and cost as the primary metrics and did not consider the issues of power or energy [17]. We also did not consider FFT processors designed specifically for executing FFT algorithms [18].

VII. Conclusion

This paper presents a DFT transform synthesis flow that captures an important range of implementation options. The synthesis flow starts from precise mathematical formulas of fast DFT algorithms and applies structural rewrite rules to impart appropriate hardware implementation decisions. The resulting annotated hardware formulas straightforwardly map to RTL netlists of efficient implementations. This synthesis flow can be coupled with the exiting Spiral formula generator to fully automate DFT design exploration and synthesis.

The formula language and the synthesis procedure presented in this paper are actually sufficient for a wider range of transforms in addition to DFT. The system, as is, can handle Walsh-Hadamard transform and multidimensional DFTs. The central limitation to supporting a broader class of transforms is in constructing cost-effective streaming implementations of the required permutations. Recent work [5] has produced very efficient solutions to this problem. Thus, we plan to continue this work on other transforms (e.g., discrete cosine transform or the DFT on real valued inputs).

Acknowledgment

This work was supported by DARPA under DOI grant NBCH-1050009 and by NSF awards ACR-0234293 and ITR/ACI-0325687.

[5]The results given for the radix-2 cases agree with our earlier work [2] which dealt specially with radix-2 horizontal-reuse Pease FFT implementations. Improvements seen in the current results are due to the recently incorporated memory-based permutation blocks.

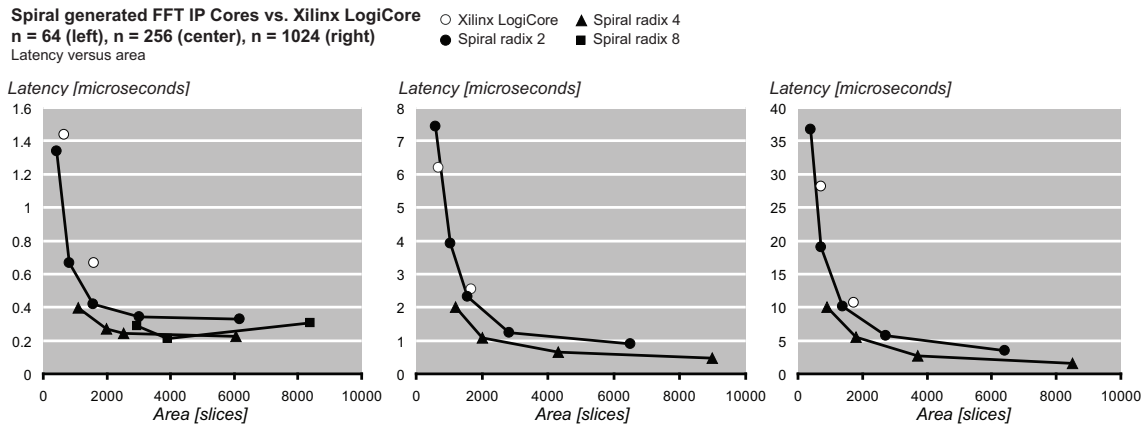


Fig. 11. Latency versus cost for horizontal-reuse implementations of DFT_{64}, DFT_{256}, and DFT_{1024} (from left to right).

n	algorithm	r	architecture	w	throughput ($1/\mu$s)	latency (μs)	cost (slices)	BRAMs	comments
64	Pease (4)	2	horiz. reuse	2	0.75	1.34	423	4	lowest cost
64	Iterative (3)	2	fully-streamed	32	42.50	0.21	19480	0	best throughput
64	Pease (4)	4	horiz. reuse	32	4.42	0.23	6052	0	lowest latency per slice
256	Pease (4)	2	horiz. reuse	2	0.13	7.44	582	2	lowest cost
256	Iterative (3)	16	fully-streamed	32	15.22	0.25	22105	64	best throughput
256	Pease (4)	4	horiz. reuse	16	0.92	1.09	2013	8	balanced cost vs latency
1024	Pease (4)	2	horiz. reuse	2	0.03	36.7	402	6	lowest cost
1024	Pease (4)	32	horiz. reuse	32	1.17	0.85	21377	124	lowest latency
1024	Pease (4)	4	horiz. reuse	32	0.37	2.72	3706	56	balanced cost vs latency

TABLE I

COMPILATION OF SELECT REPRESENTATIVE IMPLEMENTATIONS AND DESIGN CORNERS.

REFERENCES

[1] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. SIAM, 1992.

[2] G. Nordin, P. Milder, J. Hoe, and M. Püschel. Automatic generation of customized discrete Fourier transform IPs. In *Proceedings of the 42nd Annual Conference on Design Automation*, 2005.

[3] M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *ACM*, 15(2), April 1968.

[4] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[5] M. Püschel, P. A. Milder, and J. C. Hoe. Permuting streaming data using RAMs. Journal submission under preparation.

[6] Xilinx, Inc. *Xilinx LogiCore: Fast Fourier Transform v3.2*.

[7] E. H. Wold and A. M. Despain. Pipeline and parallel-pipeline FFT processors for VLSI implementations. *IEEE Transactions on Computers*, C-33(5):414–426, May 1984.

[8] S. F. Gorman and J. M. Wills. Partial column FFT pipelines. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 42(6):414–423, 1995.

[9] S. He and M. Torkelson. A new approach to pipeline FFT processor. In *Proc. International Parallel Processing Symposium*, 1996.

[10] D. Cohen. Simplified control of FFT hardware. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24(6):577–579, 1976.

[11] G. Szedo, V. Yang, and C. Dick. High-performance FFT processing using reconfigurable logic. In *Proc. Asilomar Conference on Signals, Systems and Computers*, 2001.

[12] M. Serra, P. Martí, and J. Carrabina. IFFT/FFT core architecture with an identical stage structure for wireless LAN communications. In *Proc. IEEE Workshop on Signal Processing Advances in Wireless Communications*, 2004.

[13] P. Kabal and B. Sayar. Performance of fixed-point FFT's: Rounding and scaling considerations. In *IEEE International Conference Acoustics, Speech, and Signal Processing*, volume 11, pages 221–224, April 1986.

[14] K. S. Hemmert and K. D. Underwood. An analysis of the double-precision floating-point FFT on FPGAs. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, 2005.

[15] A. Banerjee, A. Sundar Dhar, and S. Banerjee. FPGA realization of a CORDIC based FFT processor for biomedical signal processing. *Microprocessors and Microsystems*, 25(3):131–142, May 2001.

[16] M. Shaditalab, G. Bois, and M. Sawan. Self sorting radix_2 FFT on FPGAs using parallel pipelined distributed arithmetic blocks. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, 1998.

[17] S. Choi, G. Govindu, J. Jang, and V. K. Prasanna. Energy-efficient and parameterized designs for fast Fourier transform on FPGA. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, 2003.

[18] P. Kumhom, J. Johnson, and P. Nagvajara. Design, optimization, and implementation of a universal FFT processor. In *Proc. 13th IEEE ASIC/SOC Conference*, 2000.