FFT Compiler: From Math to Efficient Hardware

HLDVT Invited Short Paper

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*— This paper presents a high-level compiler that generates hardware implementations of the discrete Fourier transform (DFT) from mathematical specifications. The matrix formula input language captures not only the DFT calculation but also the implementation options at the algorithmic and architectural levels. By selecting the appropriate formula, the resulting hardware implementations (described in a synthesizable Verilog description) can achieve a wide range of tradeoffs between implementation cost and performance. The compiler is also parameterized for a set of technology-specific optimizations, to allow it to target specific implementation platforms. This paper gives a brief overview of the system and presents synthesis results.**

I. Introduction

The discrete Fourier transform (DFT) is a widely used building block in signal processing applications. The "best" algorithm and architecture to use in a hardware DFT implementation are highly dependent on the application-specific requirements in metrics such as size, throughput, latency, and tradeoffs between them. Although library cores for DFT are widely available, their fixed nature limits their ability to support application-specific customizations in the cost/performance tradeoff space. For example, the Xilinx LogiCore FFT v3.2 [1] provides just three architectures for choices in balancing cost and performance.

In this work, we describe a flexible DFT generation framework that compiles a matrix formula for a DFT algorithm into a synthesizable Verilog description of an efficient hardware implementation. By mathematically expressing algorithms and architectural parameters (such as parallelism) in the same framework, we are able to generate automatically a wide range of implementations to meet user-specified high-level design preferences.

II. Background

An n point discrete Fourier transform (DFT_n) is the matrix-vector multiplication

$$y = \mathrm{DFT}_n\, x, \quad \mathrm{DFT}_n = [e^{-2\pi jk\ell/n}]_{0\le k,\ell<n}, \quad j = \sqrt{-1}.$$

The n point complex input and output vectors are x and y, and DFT_n is an $n \times n$ complex matrix. We consider two-power values for n (i.e., $n = 2^k$).

Computation of the DFT by direct matrix-vector multiplication requires $O(n^2)$ operations. Well-known fast algorithms, called fast Fourier transforms (FFTs) reduce the operation count to $O(n \log n)$. We view an FFT algorithm as a factorization of DFT_n into a product of structured sparse matrices. We compactly express their structure using the Kronecker (or tensor) formalism [2]. The Kronecker product is

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

Of particular interest is $I_m \otimes A_n$, which is a block diagonal $nm \times nm$ matrix with A_m along the diagonal:

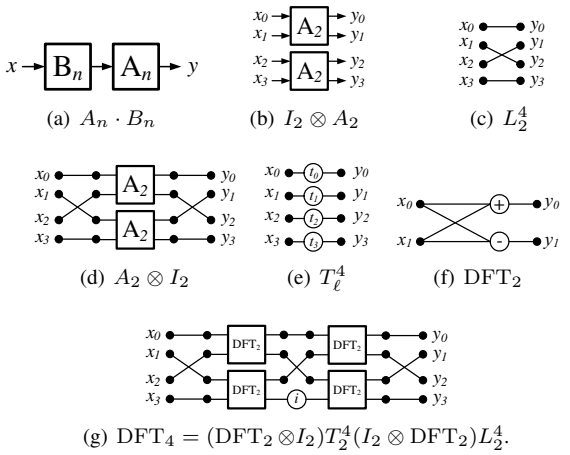
$$I_m \otimes A_n = \begin{bmatrix} A_n & & \\ & \ddots & \\ & & A_n \end{bmatrix}.$$

Using this notation, we connect FFT algorithms (i.e., procedures for computing the DFT) with explicit matrix formulas that represent the algorithm's computation. In this work, we consider two fast Fourier transform algorithms:

$$\begin{aligned} \mathrm{DFT}_{r^k} &= L^{r^k}_r \Big(\prod_{\ell=0}^{k-2} (I_{r^{k-1}} \otimes \mathrm{DFT}_r)\, T^n_\ell \\ &\quad (I_{r^\ell} \otimes L^{r^{k-\ell}}_{r^{k-\ell-1}})(I_{r^{\ell+1}} \otimes L^{r^{k-\ell-1}}_r)\Big) \qquad (1) \\ &\quad (I_{r^{k-1}} \otimes \mathrm{DFT}_r)\, R^{r^k}_r \\ \mathrm{DFT}_{r^k} &= \left(\prod_{\ell=0}^{k-1} L^{r^k}_r\, (I_{r^{k-1}} \otimes \mathrm{DFT}_r)\, T^n_\ell\right) R^{r^k}_r \qquad (2) \end{aligned}$$

In the above, multiplying a vector by a permutation matrix L^n_m corresponds to a applying a stride-by-m permutation to the vector. The "twiddle" matrix T is a diagonal matrix that is characteristic of DFTs. Multiplying a vector by T^n_ℓ scales each element of the vector by the corresponding element on T^n_ℓ's diagonal.

Equation (1) is a variant of the Iterative Cooley-Tukey FFT, and equation (2) is known as the Pease FFT. Both are iterative algorithms parameterized for vectors of length r^t (where r is known as the radix). Both algorithms are highly regular, making them well-suited for practical hardware implementations (as we will discuss in Section III).

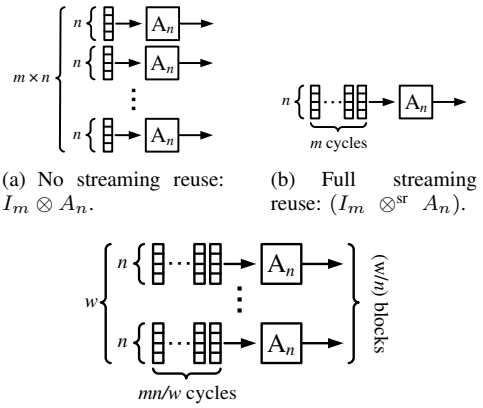


(a) $A_n \cdot B_n$ (b) $I_2 \otimes A_2$ (c) L^4_2

(d) $A_2 \otimes I_2$ (e) T^4_ℓ (f) DFT_2

(g) $\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes I_2) T^4_2 (I_2 \otimes \mathrm{DFT}_2) L^4_2$.

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



(a) No streaming reuse: $I_m \otimes A_n$.

(b) Full streaming reuse: $(I_m \otimes^{\mathrm{sr}} A_n)$.

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

Fig. 2. Examples of streaming reuse.

III. Formula-Driven Compilation

Matrix formulas can be implemented as combinational logic directly. Figure 1 shows examples of mappings from matrix formula constructs to their corresponding combinational implementations. By combining these primitives, we can build the combinational datapaths corresponding to equations (1) and (2). (A small DFT_4 example is given in Figure 1(g).)

However, as the size of the DFT grows, a combinational datapath quickly becomes prohibitively expensive. In order to build practical implementations, we must exploit regularity in the DFT computation to reuse portions of the datapath in a sequential fashion.

As an overview, our automated design flow from formula to hardware is comprised of three major stages:

Formula Generation. First, based on the desired transform size, the formula generation stage selects an algorithm and represents it as a formula. This task is completed by filling in the appropriate parameterization (i.e., r^t) into Equation (1) or (2). We consider this to be an "algorithmic" formula, because it contains a description of the procedure, but it does not yet explicitly describe the architecture for implementing it.

Formula Optimization. Next, in the formula optimization stage, high-level resource preferences and architectural parameters are provided along with the algorithmic formula. This portion of the flow produces a new formula that describes both the algorithm and the architecture that will perform it. By formally representing architectural parameters within the matrix formula, we enable a high-level restructuring of the algorithmic formula to include an implicit specification of architecture. Once this step of the design flow has completed, the architecture is completely specified, and can be directly translated into a hardware design.

RTL Generation. The final portion of our design flow maps the architectural formula into register-transfer level (RTL) Verilog. This mapping is done in a straightforward manner; this stage simply translates from an abstract representation into the RTL abstraction.

IV. Formula Optimization

In this paper, we expand on the details of the formula optimization stage with a focus on the formula-level representation of architectural parameters. This is the most important concept in our framework—it is the key to generating designs across a wide cost/performance tradeoff space. Discussion of the complete framework can be found in [3].

The goal in this stage is to exploit the high degree of regularity exhibited by algorithms (1) and (2) through datapath reuse. By controlling the parameters of reuse at the formula-level, it is possible to control the tradeoff between cost and performance of an implementation at a fine granularity. Currently, we focus on two types of reuse architecture.

Streaming. A combinational interpretation of the matrix $I_m \otimes A_n$ leads to m parallel instances of A_n. Alternatively, we can choose to interpret this matrix as *reuse across time* where data elements stream through a single instance of A_n over m cycles. We call this *streaming reuse*. To distinguish between the two possible interpretations, we introduce a tagged tensor symbol: $\otimes^{\mathrm{sr}}$. Now, $I_m \otimes A_n$ indicates parallelism in space (Figure 2(a)), and $I_m \otimes^{\mathrm{sr}} A_n$ indicates streamed reuse of A_n (Figure 2(b)). In general, we have w/n parallel instances of A_n, and the $m \times n$ data elements stream w per cycle over mn/w cycles. This architecture (as shown in Figure 2(c)) would be written as

$$I_{mn/w} \otimes^{\mathrm{sr}} (I_{w/n} \otimes A_n).$$

Horizontal reuse. If a series of identical structures is needed (e.g., $A_n A_n A_n A_n$ or simply $\prod_{\ell=0\ldots3} A_n$), rather than duplicating A_n, we can iteratively reuse a single instance of A_n. We refer to this as *horizontal reuse*. We distinguish this interpretation of repetition by the annotation $\prod^{\mathrm{hr}}$, which indicates the inner computation kernel is to be reused iteratively. When horizontal reuse is applied to a kernel block, its output feeds back to the input, and a control signal must be generated to determine whether the block receives the recirculating output values or a new set of inputs (at the start of a new computation).

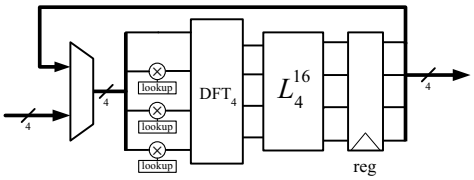


Fig. 3. Horizontal-reuse Pease DFT'_{16} with a streaming width of 4.

An important consideration with this type of reuse is that the iterated kernel cannot vary with the iteration. For example, $\prod_{\ell=0\ldots3}(L^{16}_8(I_8 \otimes \mathrm{DFT}_2))$ can be reused this way, but $\prod_{\ell=0\ldots3}(L^{16}_{2^\ell}(I_8 \otimes \mathrm{DFT}_2))$ cannot, because the stride of permutation $L^{16}_{2^\ell}$ changes in each iteration. We consider one exception to this rule: the parameter ℓ in twiddle matrices T^n_ℓ, because the operation performed (multiplication by a constant) remains the same; only the constants (stored in a lookup table) change. The Pease FFT (2), for example, is well-suited for horizontal reuse because only T^n_ℓ changes from one iteration to the next.

Example. A horizontally-reused radix 4 Pease FFT of size 16 with a streaming width of 4 is

$$\mathrm{DFT}'_{4^2} = \prod_{\ell=0\ldots1}^{\mathrm{hr}} L^{16}_4(I_4 \otimes^{\mathrm{sr}} \mathrm{DFT}_4)T^{16}_\ell.$$

The corresponding datapath is shown in Figure 3. This DFT implementation assumes that input data arrives in radix-4 bit-reversed order.

V. Evaluation

The design flow described in this paper is implemented in the following way: the formula-level portions are implemented inside the Spiral generation framework [4], and the RTL is generated by a standalone program written in Java. Designs are synthesized and place-and-routed with Xilinx ISE 8.1, targeting a Xilinx Virtex-II Pro FPGA (XC2VP100-6). In this evaluation, real and imaginary data each have a 16 bit fixed point data format (although the tool can support arbitrary fixed point bitwidths). All designs in this evaluation use bit reversed input data ordering. To provide a reference, we compare to the corresponding cores from the Xilinx LogiCore Fast Fourier Transform library v3.2 [1]. Xilinx provides cores at three cost/performance tradeoff points: a streaming core (based upon [5], and radix 2 and 4 horizontal reuse cores.

To evaluate the range of tradeoffs between cost and performance, we generate horizontal-reuse architectures with equation (2), and non-horizontal-reuse architectures with equation (1). For each architecture choice, we also consider multiple designs based on different radices and streaming widths. Figure 4 shows the Pareto-optimal set of design points for DFT_{256}. (Results for other transform sizes are qualitatively similar.) In this graph, the y-axis indicates the throughput performance as "gap" (in μs, the steady-state time between the starts of transform calculations). The x-axis is the area (in slices). The points closest to the origin have the lowest

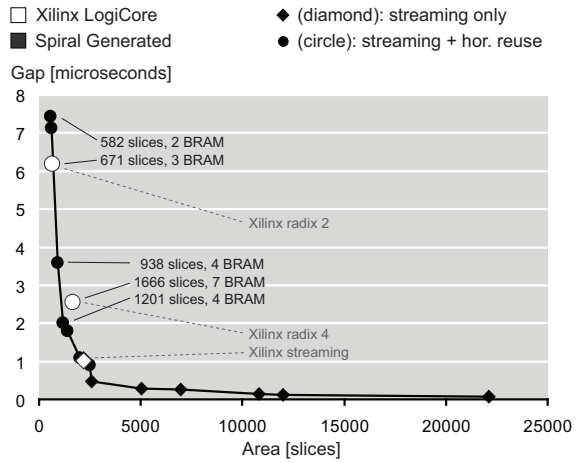


Fig. 4. Performance (gap) versus area (slices) for generated DFT_{256} cores (in black, Pareto-optimal points only) and Xilinx LogiCore library cores (in white).

cost and highest performance. In this plot, we use a diamond shape if a design does not employ horizontal reuse; we use a circle if it does. The three Xilinx LogiCore designs are marked in white. For a few data points, we also label the number of hard memory macros consumed on the FPGA. We see that our generated designs achieve a very wide range of tradeoff and can match the efficiency of the LogiCore library at comparable cost-performance tradeoff regions.

VI. Conclusions

In this paper, we presented a flexible system for generating hardware cores for computing the discrete Fourier transform. We described the matrix formula framework used, including the representation of architectural parameters. We evaluated our generated designs on a Xilinx Virtex-II Pro FPGA to show the range and competitiveness of our generated designs. The underlying formula-based framework is extensible to support other linear DSP transforms such as the discrete cosine transform, discrete wavelet transforms, and their multi-dimensional variants. Although we reported on fixed-point datapaths in the evaluation, our RTL generation stage can also support floating point formats.

References

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

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

[3] Peter A. Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Discrete Fourier transform compiler: From mathematical representation to efficient hardware. Technical Report CSSI 07-01, Center for Silicon System Implementation, Carnegie Mellon University, 2007.

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