GENERATING HIGH PERFORMANCE PRUNED FFT IMPLEMENTATIONS

Franz Franchetti and Markus Püschel
Carnegie Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA, United States

ABSTRACT

We derive a recursive general-radix pruned Cooley-Tukey fast Fourier transform (FFT) algorithm in Kronecker product notation. The algorithm is compatible with vectorization and parallelization required on state-of-the-art multicore CPUs. We include the pruned FFT algorithm into the program generation system Spiral, and automatically generate optimized implementations of the pruned FFT for the Intel Core2Duo multicore processor. Experimental results show that using the pruned FFT can indeed speed up the fastest available FFT implementations by up to 30% when the problem size and the pattern of unused inputs and outputs are known in advance.

Index Terms— Discrete Fourier transforms, Vector processing, Multiprocessing, Software performance

1. INTRODUCTION

The discrete Fourier transform (DFT) and it's fast algorithms (fast Fourier transforms, FFTs) is an ubiquitous tool in signal processing and the computational sciences. Thus, optimization of its implementation is crucial. In many applications it is either known in advance that a part of the input data of the DFT is zero or some outputs will be discarded. In this situation, applying the *pruned FFT* allows for a reduction in the operation count by discarding superfluous operations like adding values known to be zero.

Current commercial off-the-shelf (COTS) computer systems are very powerful but increasingly hard to program. Systems based on the latest Intel or AMD microprocessors have a theoretical peak performance that a decade ago was the domain of supercomputers. Microprocessors like the Intel Core2Duo or AMD's Opteron derive their computational power from architectural features like deep memory hierarchies (up to three levels of cache), vector instruction set extensions (e. g., SSE, AltiVec), and multiple processor cores (dualcore or quadcore CPUs).

While these latest architectures are incredible powerful, the development of high performance software has become a nightmare as programs with the same operations count may run at vastly different speeds. Not fully taking advantage of all architectural enhancements often means to slow down one's program by 10 to 100 times. In particular, saving operations at the expense of program complexity may not translate into speed-up but produce a slow down. This poses an enormous challenge when one tries to realize a speed-up from the lower operation count of pruned FFTs.

Contribution. In this paper we derive a general radix pruned FFT algorithm expressed in the Kronecker product formalism, applicable for block-zero patterns known in advance. We show that the pruned FFT can indeed speed up the fastest available FFT implementation, taking full advantage of multiple processor cores, vector instructions, and the memory hierarchy. For instance, when 7/8 of the inputs are known to be zero (as in 8-fold upsampling for interpolation purposes) we gain a 30% speed-up over the fastest un-pruned FFT implementation.

Related work. FFT pruning is well-studied [1, 2, 3], but traditionally pruned FFT algorithms are not expressed in the Kronecker product formalism. The goal in previous work usually was to minimize the operations count to the extent possible, albeit at the cost of increasing the implementation complexity of the algorithm. Recently, [4] investigates the fast implementation of partial FFTs on processors that support software pipelining.

For this reason, no modern high-performance FFT library like Intel's IPP and MKL, AMD's ACML, or FFTW [5] offers support for pruned FFT computation. To highlight the challenge, consider a remark by the authors of FFTW who state in the FFTW manual[1] that pruning only becomes beneficial if 99% of the inputs are known to be zero. We show in this paper that pruning has the potential to speed up high performance FFT implementations when many more inputs are non-zeros.

Organization. The paper is organized as follows. Section 2 introduces the Kronecker product formalism and the fast Fourier transform. In Section 3 we introduce the pruned FFT and derive the general radix Cooley-Tukey FFT algorithm for pruned FFT using the Kronecker product formalism. Our derived pruned FFT algorithm is experimentally evaluated in Section 4 and we offer conclusions in Section 5.

2. BACKGROUND

Discrete Fourier transform. Computing the discrete Fourier transform (DFT) of an input signal x of length N is equivalent to the matrix-vector multiplication $y = \mathrm{DFT}_N\, x$, where

$$\mathrm{DFT}_N = [\omega_N^{k\ell}]_{0 \le k,\ell < N}, \quad \omega_N = e^{-2\pi j/N}.$$

Fast Fourier transforms. Various fast Fourier transform algorithms (FFTs) are available that enable the computation of the DFT with $O(N \log N)$ operations for all sizes N. The most important one is the Cooley-Tukey FFT. It can be expressed as a factorization of the DFT_N into a product of structured sparse matrices provided that $N = mn$ factorizes. Using the Kronecker, or tensor product notation [6] this FFT is given by

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) D_{m,n} (I_m \otimes \mathrm{DFT}_n) L^{mn}_m. \qquad (1)$$

In (1), the *stride permutation* matrix L^{mn}_m permutes the input vector as

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

If x is viewed as an $n \times m$ matrix, stored in row-major order, then L^{mn}_m performs a transposition of this matrix. Further, I_n is the $n \times n$ identity matrix, and the *tensor product* is defined as

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

This work was supported by NSF through awards 0325687, 0702386, by DARPA through the DOI grant NBCH1050009 and the ARO grant W911NF0710416, and by Intel and Mercury Computing.

[1] `http://www.fftw.org/pruned.html`

$D_{m,n}$ is a diagonal matrix containing the so-called twiddle factors. Another important construct is the canonical basis vector of $\mathbb{C}^N$ with "1" at the ith location and "0" at all positions, denoted by e_i^N.

Block sequences. To describe zero-patterns we introdiuce the concept of block sequences. Let σ be an ordered sequence of integers σ_i with $0 \leq \sigma_i < N$. We denote the number of elements σ_i in σ by $|\sigma|$. We use the following short-hand notation for sequences with block structure: Let $\sigma = \langle \sigma_i \rangle_{0 \leq i < |\sigma|} \subset \{0, \ldots, N-1\}$ be an ordered sequence and k a positive integer. Then we define the ordered sequence

$$\sigma \otimes k = \langle k\sigma_i, k\sigma_i + 1, \ldots, k\sigma_i + k - 1 \rangle_{0 \leq i < |\sigma|}.$$

For instance, let $\sigma = \langle 0, 1, 3 \rangle \subset \{0, 1, 2, 3\}$ and $k = 2$. Then,

$$\sigma \otimes k = \langle 0, 1, 2, 3, 6, 7 \rangle \subset \{0, \ldots, 7\}.$$

Zero-padding a vector. Starting from a vector $x \in \mathbb{C}^{|\sigma|}$ we obtain a zero-padded vector $y \in \mathbb{C}^N$, with the non-zero entries of y being the entries of x scattered to the positions σ_i, and all other entries of y being zero. Formally, we multiply x with a scatter matrix $\mathrm{S}_\sigma \in \mathbb{C}^{N \times |\sigma|}$,

$$y = \mathrm{S}_\sigma \, x \quad \text{with} \quad \mathrm{S}_\sigma = \left[e^N_{\sigma_0} | e^N_{\sigma_1} | \ldots | e^N_{\sigma_{|\sigma|-1}} \right]. \tag{2}$$

Dropping vector elements. Conversely, we obtain the vector $y \in \mathbb{C}^{|\sigma|}$ being the subset of vector elements of $x \in \mathbb{C}^N$ at the positions σ_i by dropping all elements not at positions σ_i. This operation is formally described by multiplying the vector x by a gather matrix,

$$y = \mathrm{G}_\sigma \, x \quad \text{with} \quad \mathrm{G}_\sigma = \mathrm{S}_\sigma^\top. \tag{3}$$

The operator $(.)^\top$ denotes matrix transposition.

Formula manipulation. To derive new variants of algorithms, we apply formula identities [6] like

$$(A_m \otimes I_n) L^{mn}_m = L^{mn}_m (I_n \otimes A_m), \tag{4}$$

where A_m is an arbitrary $m \times m$ matrix. For instance, applying (4) to (1) leads to the so-called four-step variant of the Cooley-Tukey FFT algorithm,

$$\mathrm{DFT}_{mn} = (\mathrm{DFT}_m \otimes I_n) D_{m,n} L^{mn}_m (\mathrm{DFT}_n \otimes I_m). \tag{5}$$

We will also apply the distributive law for the tensor product,

$$(AB) \otimes C = (A \otimes C)(B \otimes C). \tag{6}$$

The following identity shows the connection between block sequences and matrix tensor products,

$$\mathrm{S}_{\sigma \otimes k} = \mathrm{S}_\sigma \otimes I_k, \tag{7}$$

which can be easily proved by inspection.

3. GENERAL RADIX PRUNED FFT ALGORITHM

In this section we derive the Kronecker product formulation of a general radix Cooley-Tukey pruned FFT algorithm for input or output pruning, as well as for combined input-and-output pruning.

Input pruned DFT. We now define the *input pruned DFT* as a transform that is a variant of the DFT. Assume $x \in \mathbb{C}^N$ is a vector with some entries known to be zero, i.e., $x = \mathrm{S}_\sigma \, x'$. ($x'$ are the non-zero entries of x.) The pruned DFT is the transform that computes the DFT of x from its non-zero elements x'. Formally, we write it as the matrix-vector multiplication

$$y = \mathrm{PDFT}^\sigma_N \, x' \quad \text{with} \quad \mathrm{PDFT}^\sigma_N = \mathrm{DFT}_N \, \mathrm{S}_\sigma \, . \tag{8}$$

PDFT^σ_N is a matrix of $N \times |\sigma|$ complex roots of unity, consisting of the columns σ_i of DFT_N.

Output pruned DFT. Dropping some of the outputs of the DFT computation (*output pruning*) is expressed by formally transposing the input pruned DFT. Assume $y \in \mathbb{C}^N$ is a vector where some elements are to be dropped (the elements kept are denoted by y'), which is expressed by $y' = \mathrm{G}_\sigma \, y$. The output pruned DFT is the transform that computes the DFT of $x \in \mathbb{C}^N$ and then drops the superfluous elements to yield y'. It is obtained using the transpose of the input pruned DFT,

$$y' = (\mathrm{PDFT}^\sigma_N)^\top x \quad \text{with} \quad (\mathrm{PDFT}^\sigma_N)^\top x = \mathrm{G}_\sigma \, \mathrm{DFT}_N \, .$$

Input and output pruned DFT. We define the *simultaneously input and output pruned DFT* as the combination of the input and the output pruned DFT. Some elements of the input vector x are known to be zero ($x = \mathrm{S}_\sigma \, x'$) and some of the elements of the output vector y are to be dropped ($y' = \mathrm{G}_\tau \, y$). The simultaneously pruned DFT computes the kept output elements y' from the non-zero input elements x'. Formally, we write it as the matrix-vector multiplication

$$y' = \mathrm{PPDFT}^{\tau,\sigma}_N \, x' \quad \text{with} \quad \mathrm{PPDFT}^{\tau,\sigma}_N = \mathrm{G}_\tau \, \mathrm{DFT}_N \, \mathrm{S}_\sigma \, . \tag{9}$$

Recursive pruned Cooley-Tukey FFT algorithm. Let $N = kmn$ and $\sigma \subset \{0, \ldots, n-1\}$. We now derive a variant of the general radix Cooley-Tukey FFT recursion (1) for the input or output pruned DFT,

$$\mathrm{PDFT}^{\sigma \otimes km}_{kmn} \quad \text{and} \quad (\mathrm{PDFT}^{\sigma \otimes km}_{kmn})^\top.$$

Substituting (5) in (8) leads to

$$\mathrm{PDFT}^{\sigma \otimes km}_{kmn} = (\mathrm{DFT}_m \otimes I_{kn}) D L^{kmn}_m (\mathrm{DFT}_{kn} \otimes I_m) \, \mathrm{S}_{\sigma \otimes km}$$

with $D = D_{m,kn}$. Applying identities (7) and (6) pushes the scatter matrix $\mathrm{S}_{\sigma \otimes km}$ down into the recursion,

$$\mathrm{PDFT}^{\sigma \otimes km}_{kmn} = (\mathrm{DFT}_m \otimes I_{kn}) D L^{kmn}_m (\mathrm{DFT}_{kn} \, \mathrm{S}_{\sigma \otimes k} \otimes I_m).$$

Finally, applying (8) in the recursion yields the recursive general radix Cooley-Tukey FFT algorithm for input pruning,

$$\mathrm{PDFT}^{\sigma \otimes km}_{kmn} = (\mathrm{DFT}_m \otimes I_{kn}) D L^{kmn}_m (\mathrm{PDFT}^{\sigma \otimes k}_{kn} \otimes I_m). \tag{10}$$

Formal transposition of (10) yields the recursive general radix Cooley-Tukey FFT algorithm for output pruning.

Let $N = k\ell mn$, $\sigma \subset \{0, \ldots, n-1\}$, and $\tau \subset \{0, \ldots, m-1\}$. We now derive a variant of the general radix Cooley-Tukey FFT recursion (1) for the simultaneously input and output pruned DFT,

$$\mathrm{PPDFT}^{\tau \otimes k\ell n, \sigma \otimes k\ell m}_{k\ell mn}.$$

We substitute (5) in (9) and apply the identities (6)–(8) as above. This yields the simultaneously input and output pruned general radix Cooley-Tukey FFT algorithm,

$$\begin{aligned} \mathrm{PPDFT}^{\tau \otimes k\ell n, \sigma \otimes k\ell m}_{k\ell mn} \quad = \quad & ((\mathrm{PDFT}^{\tau \otimes \ell}_{\ell m})^\top \otimes I_{kn}) D_{\ell m, kn} \\ & \cdot L^{k\ell mn}_{\ell m} (\mathrm{PDFT}^{\sigma \otimes k}_{kn} \otimes I_{\ell m}). \end{aligned} \tag{11}$$

Base cases. (10) and (11) perform recursive pruning of the FFT algorithm. They require a pattern of non-zero elements described by

block sequences of composite block length. The block length gets factored by the recursion steps. Once the block length becomes one the rules are no longer applicable. Under realistic assumptions the corresponding DFT sizes are small and can be optimized similarly to the pruning applied by the FFTW codelet generator `genfft` [7].

For small enough N, PDFT_N^σ and $\mathrm{PPDFT}_N^{\tau,\sigma}$ can be implemented using a single basic block of straight-line code. Inside the basic block one can apply constant propagation and dead code elimination (either by hand or using an automatic tool) to obtain an optimized pruned FFT implementation that does not perform unnecessary operations.

Cost analysis. Computing $y = \mathrm{DFT}_N\, x$ has a cost of $O(N \log N)$, and the radix-2 FFT has an asymptotic cost of $5N \log_2 N$ [6]. For $|\sigma| = k$ computing $y = \mathrm{PDFT}_N^\sigma\, x$ has a cost of $O(N \log k)$ [3]. The exact operations count depends on the chosen recursion, which is restricted by the block structure of σ, and is hard to determine by a closed formula. Nevertheless, pruning reduces only the $\log N$ term to $\log k$ and thus reduces the operations count only modestly. For instance, if 7/8 of the inputs of a signal with 1,024 samples are known to be zero ($N = 1,024$ and $k = 128$), using the pruned FFT reduces the operations count by 30%.

Automatically generating pruned DFTs. Spiral[2] is a program generation and optimization system for transforms including the DFT [8]. For a given a DFT of size N, Spiral expands DFT_N recursively using (1) or other FFT algorithms until base cases ($N = 2$) are reached. The resulting formula (tensor product expression) is further optimized as described in [9] and then translated into C code. Based on the runtime of the obtained code, Spiral changes the recursive expansion (e.g., by choosing different factorizations $N = mn$) and repeats the process. This search eventually produces an implementation tuned to the given computer. Spiral generates highly optimized SIMD vectorized FFT implementations based on the short vector Cooley-Tukey FFT algorithm [10]. On multicore CPUs, Spiral generates multi-threaded implementations that provide speed-up starting with very small FFT sizes [11].

We added the general radix pruned FFT algorithm we derived in Section 3 to Spiral's algorithm database in a way that is compatible with vectorization and parallelization. This enables us to let Spiral automatically generate very fast pruned FFT implementations and evaluate them for practically relevant parameters. By adding the pruned FFT algorithm to Spiral, we enable Spiral to automatically investigate the trade-off between FFT pruning, data locality, multicore parallelization, and SIMD vectorization. It automatically finds the algorithm that matches the architecture and zero-pattern best.

4. EXPERIMENTAL RESULTS

Test setup. We evaluate the performance of our automatically generated single-precision pruned DFT implementations on a 2.66 GHz Intel Core2Duo running Windows XP in 32-bit mode. All codes were compiled with the Intel C++ compiler 10.1 with options "/O3 /QxT," which enables support for the SSSE3 instruction set. SSSE3 instructions are explicitly inserted using the intrinsic function interface provided by the Intel C++ compiler. We built our multi-threaded implementations using the threading interface provided by the VisualStudio C libraries. All implementations are specialized to the zero-pattern and the problem size.

All graphs in Fig. 1 show performance in *pseudo Gflop/s*, computed as $5N \log_2 N/\text{runtime}$. This states the performance a radix-2 FFT would have to achieve to reach the same runtime as the measured implementation for the given problem size (and extrapolates the measure for non-two-powers). The measure is normalized inverse runtime and takes into account both the speed and the reduction of arithmetic operations, allowing to compare vastly different FFT algorithms, both pruned and unpruned.

Baseline implementation: DFT. As first experiment we establish that our baseline DFT implementation is as fast as the fastest available DFT implementations. Fig. 1 (a) shows the performance of Spiral-generated single-threaded, vectorized (pink line, squares) and multi-threaded, vectorized (red line, circles) DFT implementations, compared to the single-threaded, vectorized implementation of FFTW 3.1.2 (blue line, diamonds), Intel MKL 9.0 (green line, x), and Numerical Recipes in C [12] (only two-powers, black crosses).

The single-threaded Spiral-generated DFT implementation is slightly faster than its FFTW equivalent and about equal to single-threaded MKL for two-powers. For non-two-power sizes MKL shows a drastic performance drops. Neither MKL nor FFTW are sped up by the second core for the tested problem sizes. Spiral-generated multi-threaded code starts taking advantage of the second core from size 2,048 on and achieves up to 75% speed-up over its single-threaded counterpart. Numerical Recipes is more than 20 times slower than the fastest code on this machine, even though it has only about a 20% higher operations count.

Input pruning. Next, we show that a pruned FFT algorithm indeed can speed up the fastest available DFT implementation, and that the speed-up behaves as expected. From the cost analysis in Section 3 we conclude that one can expect 5% to 30% speed-up, with smaller speed-up values for larger problem sizes or less zeros. We show the performance behavior of 4 different zero-patterns, with 1/2, 3/4, 7/8, and 15/16 of the inputs known to be zero. The patterns are derived from practical applications like interpolation or downsampling: either the center data is zero or the non-zero data is concentrated at the beginning or center of the data vector.

In Fig. 1 (b) we compare single-threaded, vectorized, Spiral-generated DFT to Spiral-generated pruned DFT implementations, with varying degree of zeros. (The baseline "Spiral DFT" is the line "Spiral DFT SSE" from Fig. 1 (a).) Output pruned DFTs with the same zero-pattern behave equivalently to the input-pruned implementations. FFT pruning gives a pronounced speed-up for sizes up to 1,024, and then the speed-up drops significantly. This is due to the fact that up to 1,024 Spiral implements the DFT with 2 stages (recursion 32×32, DFT_{32} as as base case), while for larger sizes it requires more than 2 stages. As predicted, the speed-up is highest for the patterns with the largest numbers of zeros (15/16), and the performance approaches the unpruned DFT once only half of the inputs are known to be zero.

In Fig. 1 (c) we investigate the behavior of pruned DFTs in a multi-threaded setting. We perform the exact same experiments as shown in Fig. 1 (b), but ask Spiral to generate vectorized parallel implementations that take advantage of both processor cores and the SSSE3 instruction set. (The baseline "Spiral DFT" is the line "Spiral DFT SSE+SMP" from Fig. 1 (a).) We see that pruning indeed is able to speed up the fastest DFT implementations, which take advantage of SIMD vector instructions and multiple cores. As in Fig. 1 (a), the second processor starts providing speed-up around 2,048. The speed-up obtained through pruning is similar to what we observed in the single processor results in Fig. 1 (b).

Input and output pruning. Finally, we investigate the impact of simultaneously pruning the input and output of a DFT. Fig. 1 (d) compares the performance of an un-pruned DFT to two scenarios in which both input and output patterns are known. In both cases only 1/4 of the inputs are non-zero, and 1/8 or 1/16 of the outputs are non-zero respectively. We only evaluate the performance of SSSE3

[2] `www.spiral.net`

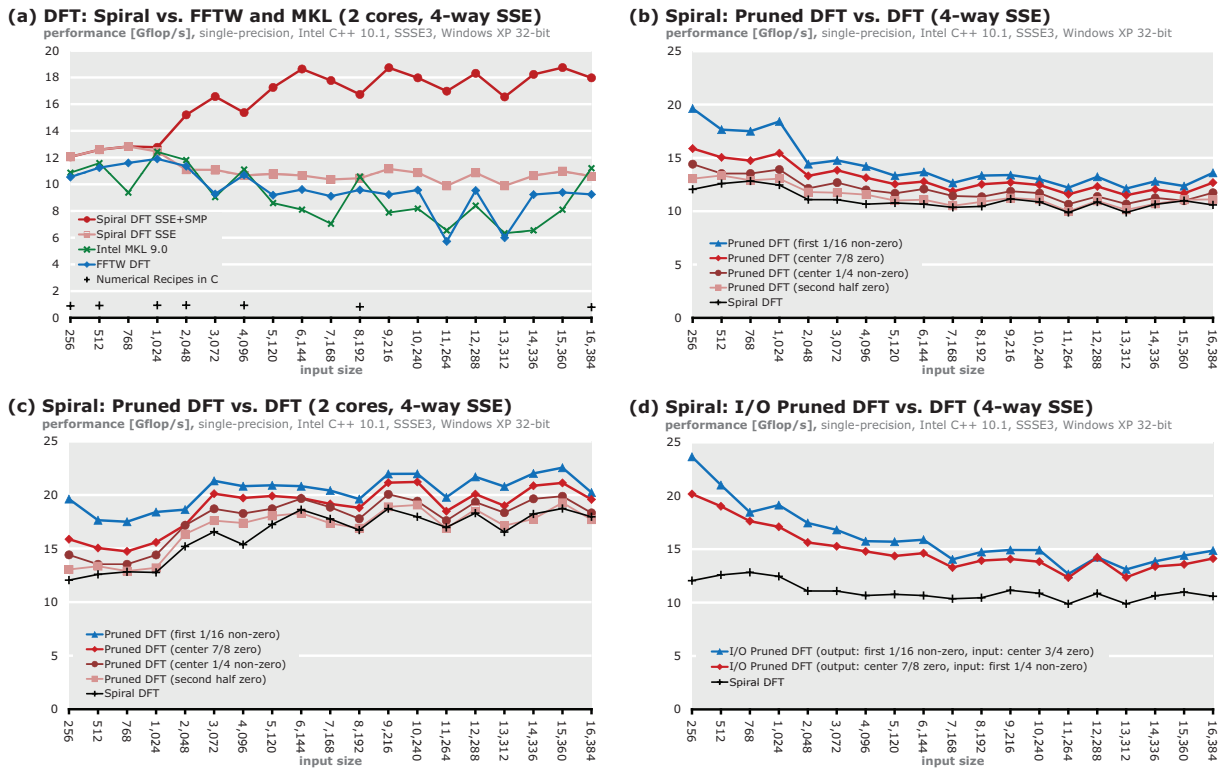

Fig. 1. Performance of DFT and pruned DFT on a 2.66 GHz Intel Core2Duo. Higher is better.

code on a single core. The obtained speed-up is higher than the input pruned-only speed-up, but has the same general behavior. For small sizes the performance gain is very pronounced, peaking at almost 80% speed-up.

5. CONCLUSION

In this paper we derive a Kronecker product formulation of the general radix pruned Cooley-Tukey FFT algorithms for input or output pruning and simultaneous input and output pruning. The formulation is compatible with formal parallelization and vectorization and automatic program generation and optimization. It supports block-zero patterns that are known in advance. We include the algorithms into the program generation system Spiral and automatically generate pruned FFT implementations that provide speed-up over the fastest un-pruned FFT implementations available. This shows that FFT pruning not only reduces the operations count but can lead to considerable speed-up on modern computer systems.

6. REFERENCES

[1] J. D. Markel, "FFT pruning," *IEEE Trans. Audio Electroacoust.*, vol. 19, no. 4, pp. 305–311, 1971.

[2] T. Sreenivas and P. Rao, "FFT algorithm for both input and output pruning," *IEEE Trans. Acoustics, Speech and Signal Processing*, vol. 27, no. 3, pp. 291–292, 1979.

[3] H. V. Sorensen and C. S. Burrus, "Efficient computation of the DFT with only a subset of input or output points," *IEEE Trans. Signal Processing*, vol. 41, no. 3, pp. 1184–1200, 1993.

[4] Min Li, David Novo, Bruno Bougard, Liesbet Van Der Perre, and Francky Catthoor, "Generic multi-phase software-pipelined partial-fft on instruction-level-parallel architectures and sdr baseband applications," in *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, New York, NY, USA, 2008, pp. 598–603, ACM.

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

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

[7] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 1999, pp. 169–180.

[8] 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*, vol. 93, no. 2, pp. 232–275, 2005, special issue on *Program Generation, Optimization, and Adaptation*.

[9] F. Franchetti, Y. Voronenko, and M. Püschel, "Loop merging for signal transforms," in *Proc. Programming Language Design and Implementation (PLDI)*, 2005, pp. 315–326.

[10] F. Franchetti and M. Püschel, "Short vector code generation for the discrete Fourier transform," in *Proc. Int'l Parallel and Distributed Processing Symposium (IPDPS)*, 2003, pp. 58–67.

[11] F. Franchetti, Y. Voronenko, and M. Püschel, "FFT program generation for shared memory: SMP and multicore," in *Proc. Supercomputing (SC)*, 2006.

[12] W. H. Press, B. P. Flannery, Teukolsky S. A., and Vetterling W. T., *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 2nd edition, 1992.