# High Performance Synthetic Aperture Radar Image Formation On Commodity Multicore Architectures

Daniel S. McFarlin, Franz Franchetti, Markus Püschel, and José M.F. Moura

Electrical and Computer Engineering
Carnegie Mellon University

## ABSTRACT

Synthetic Aperture Radar (SAR) image processing platforms have to process increasingly large datasets under and hard real-time deadlines. Upgrading these platforms is expensive. An attractive solution to this problem is to couple high performance, general-purpose Commercial-Off-The-Shelf (COTS) architectures such as IBM's Cell BE and Intel's Core with software implementations of SAR algorithms. While this approach provides great flexibility, achieving the requisite performance is difficult and time-consuming. The reason is the highly parallel nature and general complexity of modern COTS microarchitectures. To achieve the best performance, developers have to interweave of various complex optimizations including multithreading, the use of SIMD vector extensions, and careful tuning to the memory hierarchy. In this paper, we demonstrate the computer generation of high performance code for SAR implementations on Intel's multicore platforms based on the Spiral framework and system. The key is to express SAR and its building blocks in Spiral's formal domain-specific language to enable automatic vectorization, parallelization, and memory hierarchy tuning through rewriting at a high abstraction level and automatic exploration of choices. We show that Spiral produces code for the latest Intel quadcore platforms that surpasses competing hand-tuned implementations on the Cell Blade, an architecture with twice as many cores and three times the memory bandwidth. Specifically, we show an average performance of 39 Gigaflops/sec for 16-Megapixel and 100-Megapixel SAR images with runtimes of 0.56 and 3.76 seconds respectively.

**Keywords:** Synthetic Aperture Radar, SIMD Vectorization, Multithreading, Parallel Processing, Program Generation

## 1. INTRODUCTION

Commercial off-the-shelf (COTS) systems, based on Intel or AMD multicore processors, offer high potential performance at low price. However, obtaining high performance for real-world data intensive applications like image formation in synthetic aperture radar (SAR) is very difficult. To take full advantage of commodity architecture programmers must optimize programs for the memory hierarchy and parallelize them for multiple CPU cores. Computational kernels need to be vectorized for SIMD vector instructions. Failing to perform these optimizations can result in performance losses by one or two orders of magnitude.

**Polar Format SAR Image Formation.** The Image Formation process for Polar Format Algorithm (PFA) SAR is well known. We present a summarized version here for completeness and refer the reader to more comprehensive discussions. [1–3] SAR is collected as a sequence of $m_1$ pulses constituting the cross-range dimension, each comprised of $n_1$ complex samples which constitute the range dimension. The geometry of the collected data is polar, not rectangular: the pulses are arranged in the so-called Polar Annulus and a pulse's angular orientation in the Annulus reflects the physical viewing angle at which it was sampled during the collection process. In general, the Polar Annulus is a 2D Fourier domain representation of the scene's time-domain radar reflectivity function. An inverse 2D Fourier Transform is applied to the Polar Annulus to recover the radar reflectivity function. The geometry requires a *non-uniform* inverse Fourier transform, which is implemented as interpolation from polar grid to rectangular grid followed by a standard 2D inverse Fourier transform. The choice of a suitable interpolation is crucial for achieving good accuracy and depends on the geometry of the Polar Annulus. This process is called "re-gridding" (resampling of the radar reflectivity function from a curvilinear grid to a rectangular grid). [3]

**Contribution.** The main contribution of this paper is to enable automatic program generation for polar format SAR, using the program generation framework Spiral. We first derive a formal specification of the algorithm as declarative breakdown rules in Spiral's proprietary internal formula language. [4,5] In this description we use pruned FFTs to reduce the operations count. Key to highest performance, however, is the efficient parallelization, vectorization, and tuning to the memory hierarchy. We achieve this inside Spiral through rewriting rules that perform these optimizations at a high level

{dmcfarli,franz,pueschel,moura}@ece.cmu.edu

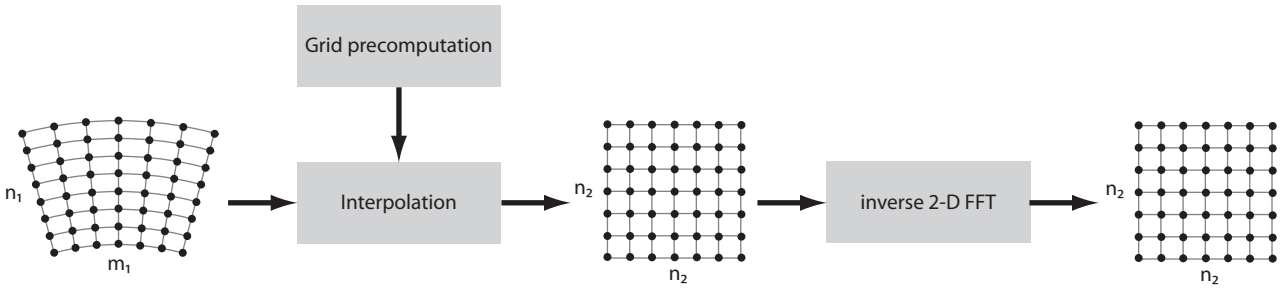



Figure 1. SAR Process Flow: The input Polar Annulus has $n_1$ samples and $m_1$ pulses. The grid is first projected onto $m_1$ pulses and then $l$ secant lines for all $n_1$ cross-range arcs. For range interpolation, each pulse is divided into $\ell$ segments of length $m$. For each segment, a forward FFT of size $m$ is performed followed by an inverse FFT of size $n$. Cross-range interpolation follows an identical sequence. Following interpolation, a 2D Inverse FFT operates on the reformatted $n_2$ x $n_2$ Cartesian grid.

of abstraction. Further optimization is achieved by Spiral's automatic search over functionally equivalent alternatives. In doing so, Spiral optimizes the compute-intensive parts of SAR collectively and holistically.

Experimental evaluation shows that the performance of our computer-generated SAR code on a commodity Intel quad-core platform is comparable to a hand-tuned implementation for specialized, Cell BE-based hardware with eight cores, [6] achieving up to 40 Gflop/s or 3.5 seconds for a 100 Megapixel image.

**Related Work.** The open literature contains many instances of High Performance SAR on FPGAs, PowerPC and ASICs.[7–9] This is unsurprising given the embedded nature of most SAR applications and the dominance of these architectures in the embedded space. Most instances of SAR on x86 focus on cluster performance and/or MATLAB performance.[3,9,10] In other cases, the algorithms or the problem sizes under investigation prevent direct comparison. [11,12] Recently, the most active area of exploration for High Performance SAR has been on the Cell BE architecture. The high aggregate memory bandwidth of the Cell in addition to its impressive theoretical peak floating point performance have made it a prominent platform in the embedded space. We chose to emulate the large problem sizes and algorithm choices of a recent effort on the Cell[6] due to their compelling performance results and the recentness of their contribution.

## 2. SAR ALGORITHM SPECIFICATION

The general PFA SAR Image Formation process described above is amenable to a wide variety of customized interpolation algorithms, 2D FFT implementations, projection techniques and geometric approximations. We now describe the specifics of the PFA SAR Image Formation algorithm that we implement, which is based on. [6] A high-level visual representation of the process is shown in Figure 1. We now discuss the blocks in greater detail.

**Grid Computation and Interpolation.** Interpolation involves a resampling of the radar reflectivity function from a curvilinear grid, the Polar Annulus, to a rectangular grid, the Cartesian grid array. Equivalently, the sample points of the function in the Polar Annulus are aligned to their nearest neighbor on an inscribed rectangular grid. Alignment requires a geometric projection of the grid lines of Polar Annulus onto the lines of the Cartesian grid array. We begin by superimposing a Cartesian grid on the Polar Annulus. Then, for each grid point $(x, y)$ in Cartesian space we find the coordinates (pulse number, sample number) of the corresponding element in the Polar Annulus. This correspondence is established with geometric projections that take into account the angular orientation of each pulse and the Cartesian distances ($dx$ and $dy$) between between grid points. There are subtleties to these projections having to do with uniform versus non-uniform spacing of Polar Annulus elements in range and cross-range. We explore these issues below but the main difficulty comes from the fact that in general the grid points have Cartesian coordinates $(x, y)$ which are real-valued whereas Polar Annulus elements have coordinates (pulse number, sample number) which are integer-valued as shown in Figure 3. Simply rounding the Cartesian coordinates may result in a sequence of Polar Annulus elements which exhibits large discontinuities. The well-known approach to avoiding discontinuities involves interpolating between elements of the Polar Annulus in both the range and cross-range dimensions. [1] Grid parameters such as the number of grid points (i.e. the size of the grid) and the $dx$, $dy$ distance between them are determined by a host of scenario and radar parameters. The grid is typically centered on the center frequency of the central pulse. An extended discussion of grid formation and placement can be found in Carrara's book. [1]

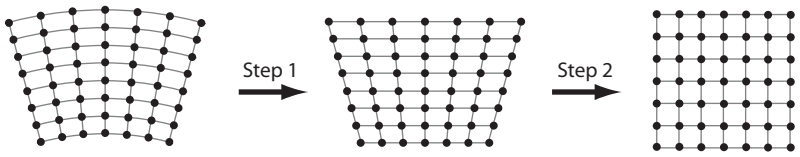



Figure 2. SAR Interpolation

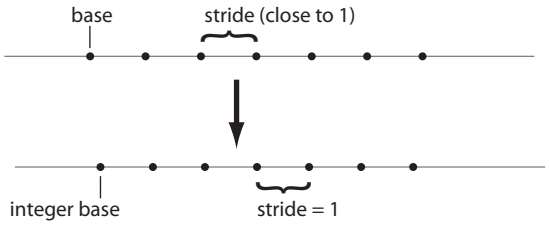



Figure 3. The projection of the real-valued indices of the Cartesian grid onto the integer-valued indices of the Polar Annulus requires resampling. We compute base and stride values for each cross-range and range line based on the projection geometry.

Once the grid is centered, we then compute the projections for the range dimension. For each pulse, $P_i$ we compute the projection of a line segment between two adjacent grid points on the central grid-vertical onto $P_i$. The length of this projection represents the stride, $s_i$ or distance along $P_i$ between any two projected grid points and is constant for a particular $P_i$. We also determine the intersection of the bottom boundary of the grid with $P_i$. The distance between this intersection and the first element of $P_i$ is known as the base, $b_i$ and is used by the interpolation algorithm to skip over elements that fall outside of the grid.

In contrast to elements in range, elements in the cross-range dimension are not uniform in spacing. This is due to the fact that cross-range elements are aligned on concentric arcs that form the Polar Annulus and are more densely packed on the initial arcs than the later arcs. We approximate the projection of points from an arc to a line with $\ell$ secants per arc. Secants provide a reasonable approximation for arcs associated with the long distance, long aperture collection geometries that we investigate here. The projection process for each secant line is nearly identical to the one describe above for a particular pulse with the major difference being that we accumulate $\ell$ $(b, s)$ pairs per arc.

**Interpolation Algorithm.** We follow the approach in;[6] a visual representation of this process is depicted in Figure 2. We first segment the data into $\ell$ segments with overlap $r$, and then upsample each segment by a factor $k$. To interpolate a segment $S_i$ of $m$ elements, we first transform the data via an FFT. Then we zero-pad the center 7/8 $^{\text{ths}}$ of the transformed data, yielding $km$ elements. Next, we perform an inverse FFT, obtaining the upsampled signal $S_i^{'}$ of $km$ elements. Following upsampling, we select elements from $S_i^{'}$ with indices of the form $b_i + js_i$ for all $j \in \{0..o_i\}$ where $o_i = \lfloor (n - b_i)/s_i \rfloor$. The selected elements are then stored contiguously in an output array called $S_i^{''}$.

The total upsampling and downsampling process (i.e. interpolation) is nearly identical for both range and cross-range with segments in range being equivalent to secants in cross-range. The major difference in cross-range interpolation is a blocking restriction imposed by vectorization. Blocking requires us to logically divide all arcs into a sequence of non-overlapping blocks. Each block contains a number of arcs equal to the vector width of the architecture (four in the case of Intel's SSE). We then make tiny adjustments to $b$ and $s$ values on a per secant basis to ensure a uniform $o$ value for all corresponding secants in a block. These adjustments are safe because of the collection geometries of the scenarios; the small coherent integration angles and large number of pulses result in only minute geometric projection differences between the corresponding secants in a block.

**2D Inverse FFT.** The specification of the 2D FFT is standard.

**Computational Cost.** The most computationally intensive portion of PFA SAR Image Formation is interpolation; the final 2D FFT is nearly an order of magnitude less demanding in terms of operations count. The high operations count for interpolation comes from the fact that we must perform two FFTs per segment/secant for each range/cross-range line. Stated more formally, and assuming $5n \log_2(n)$ operations for an $n$-point FFT, the highest-order term of the operations count for range interpolation is $5\ell m_1(m \log_2(m) + n \log_2(n))$ where $\ell$ is the number of segments per range line, $m_1$ is number of range lines, $m$ is the input segment size and $n$ is the size of the upsampled output segment. The figure is identical for the cross-range dimension with $n_1$ substituted for $m_1$. The operations count for the 2D FFT is about $10n_2^2 \log_2(n_2)$ assuming a square Cartesian grid.

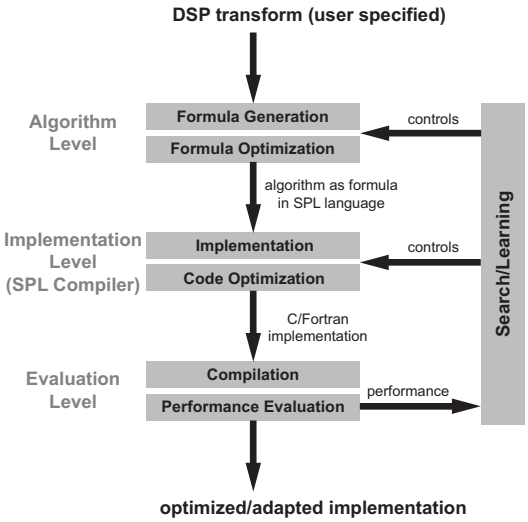



Figure 4. The program generator Spiral.

## 3. SPIRAL: A PROGRAM GENERATION FRAMEWORK

In this section we first describe Spiral, the program generation framework and system we extend to support PFA SAR. In the next section, we then guide the reader through the process by which the PFA SAR Image Formation algorithm's specification is represented in our framework. Finally, we show how this representation is formally manipulated, transformed and emitted as a highly optimized C program.

### 3.1 Spiral

Spiral is a program generator for linear transforms, most notably the discrete Fourier transform (DFT), the Walsh-Hadamard transform (WHT), the discrete cosine and sine transforms, finite impulse response (FIR) filters, and the discrete wavelet transform. The input to Spiral is a formally specified transform (e.g., DFT of size 1,024); the output is a highly optimized C program implementing the transform. Recently, we extended Spiral beyond the transform domain; [4] this paper focusses on the extension to PFA SAR.

In Spiral[5] (see Figure 4), recursive computation of larger computational kernels by smaller kernels (divide-and-conquer) is expressed using *breakdown rules*. For a given kernel, Spiral recursively applies these rules to generate one out of many possible algorithms represented as a *formula* in a language called operator language (OL). [4] This formula is then structurally optimized using a rewriting system[13] and finally translated into a C program (for computing the kernel) using a special formula compiler.[14,15] Structural optimization includes parallelization for different forms of parallelism, including multiple threads[16] and SIMD vector instructions.[17,18] The performance of the generated code is measured or estimated and fed into a search engine, which decides how to modify the algorithm. This means, the search changes the formula, and thus the code, by using dynamic programming or other search methods. Eventually, this feedback loop terminates and outputs the fastest program found in the search.

Here, we fcous on the components of the system relevant for the representation of the PFA SAR Image Formation algorithm.

**DFT and FFT.** A (linear) transform is a matrix-vector multiplication $x \mapsto y = Mx$, where $x$ is a real or complex input vector, $M$ the transform matrix, and $y$ the result. For example, for an input vector $x \in \mathbb{C}^n$, the DFT and iDFT are defined by the matrix

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

Algorithms for transforms are typically divide-and-conquer and can be viewed as factorizations of the transform into sparse, structured matrices.[19] The structure is exhibited through matrix operators like the product $\cdot$ and the Kronecker product $\otimes$ defined as

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

Then, for example, the Cooley-Tukey FFT algorithm can be written as the *breakdown rule*

$$\mathrm{DFT}_{mn} \to (\mathrm{DFT}_m \otimes \mathrm{I}_n) D_{m,n} (\mathrm{I}_m \otimes \mathrm{DFT}_n)\, \mathrm{L}_m^{nm} \tag{1}$$

with the identity matrix $\mathrm{I}_n$, the diagonal matrix $D_{m,n}$ and the stride permutation matrix $\mathrm{L}^{nm}_m$. The iDFT has an analogous rule. Using the Kronecker product, the 2D-DFT is formally defined by

$$\mathrm{DFT}_{m\times n} = \mathrm{DFT}_m \otimes \mathrm{DFT}_n\,.$$

A well-known algorithm to compute a 2D-DFT is the row column algorithm, [19]

$$\mathrm{DFT}_{m\times n} \rightarrow (\mathrm{DFT}_m \otimes \mathrm{I}_n)(\mathrm{I}_m \otimes \mathrm{DFT}_n). \tag{2}$$

Here, a first pass computes 1D DFT's along the rows, followed by a second pass of 1D DFT's along the columns; Any 1D DFT algorithm can be used.

**Pruned FFT.** Knowing that some inputs to an FFT are zero allows to reduce the operations count, since a part of the computation becomes addition by zero. This idea is used in the Pruned FFT and the Cooley-Tukey Pruned FFT algorithm. [20] The zero pattern is propagated recursively into the smaller DFT kernels, to allow a loop-based FFT implementation with reduced operations count. If the input vector of length $n$ has $k$ non-zeros then the computational cost becomes $O(n \log k)$ for the pruned DFT compared to $O(n \log n)$ for the standard DFT.

To describe the Pruned FFT algorithm we need the canonical basis vector of $\mathbb{C}^N$ with "1" at the $i$th location and "0" at all positions, denoted by $e^N_i$; to describe zero-patterns we introduce the concept of block sequences. Let $\sigma$ be an ordered sequence of integers $\sigma_i$ with $0 \le \sigma_i < N$. We denote the number of elements $\sigma_i$ in $\sigma$ by $|\sigma|$. We use the following short-hand notation for sequences with block structure: Let $\sigma = \langle \sigma_i \rangle_{0\le i<|\sigma|} \subset \{0,\ldots,N-1\}$ be an ordered sequence and $k$ a positive integer. Then we define the ordered sequence

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

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 $\sigma_i$, and all other entries of $y$ being zero. Formally, we multiply $x$ with a scatter matrix $\mathrm{S}_\sigma \in \mathbb{C}^{N\times|\sigma|}$,

$$y = \mathrm{S}_\sigma\, x \quad \text{with} \quad \mathrm{S}_\sigma = \left[e^N_{\sigma_0} | e^N_{\sigma_1} | \ldots | e^N_{\sigma_{|\sigma|-1}}\right]. \tag{3}$$

We now define the *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

$$y = \mathrm{PDFT}^\sigma_N\, x' \quad \text{with} \quad \mathrm{PDFT}^\sigma_N = \mathrm{DFT}_N\, \mathrm{S}_\sigma\,. \tag{4}$$

$\mathrm{PDFT}^\sigma_N$ is a matrix of $N \times |\sigma|$ complex roots of unity, consisting of the columns $\sigma_i$ of $\mathrm{DFT}_N$. Let $N = kmn$ and $\sigma \subset \{0,\ldots,n-1\}$. Then the recursive general radix Cooley-Tukey FFT algorithm for pruning is given by

$$\mathrm{PDFT}^{\sigma\otimes km}_{kmn} = (\mathrm{DFT}_m \otimes \mathrm{I}_{kn}) D_{m,n}\, \mathrm{L}^{kmn}_m (\mathrm{PDFT}^{\sigma\otimes k}_{kn} \otimes \mathrm{I}_m). \tag{5}$$

For problem sizes and zero patterns in this paper the pruned DFT reduces the operations count by approximately 10%–15%. The inverse pruned DFT (iPDFT) and its Cooley-Tukey FFT algorithm is defined analogously.

### 3.2 Parallelization and Vectorization

We now discuss Spiral's approach to formal parallelization and vectorization. [21] The key observation is that formula constructs can be related to properties of the target architecture. In particular, certain formula constructs can be implemented efficiently on a particular type of hardware while they are ill-suited for other types of hardware. As example, in (1) the construct

$$\mathrm{I}_m \otimes \mathrm{DFT}_n \tag{6}$$

has a perfect structure for $m$-way parallel machines. Similarly, the construct

$$\mathrm{DFT}_m \otimes \mathrm{I}_n \tag{7}$$

has a perfect structure for $n$-way vector SIMD (single instruction, multiple data) architectures. [18] However, (6) is ill-suited for vector SIMD architectures and (7) is ill-suited for parallel machines.

Further, formulas can be manipulated using algebraic identities [22] to change their structure. For instance, the identity

$$\mathrm{I}_{mn} = \mathrm{I}_m \otimes \mathrm{I}_n \tag{8}$$

breaks a identity matrix into a tensor product of identity matrices. Similar identities allow to transform vector constructs into parallel constructs and vice versa, albeit at some additional cost. The underlying general rule to flip a tensor product is

$$A \otimes B = \mathrm{L}^{mn}_m (B \otimes A)\, \mathrm{L}^{mn}_n$$

for a $m \times m$ matrix $A$ and a $n \times n$ matrix $B$.

**Optimization through rewriting.** The basic idea in Spiral's formula optimization is to rewrite a generated formula into another formula that has a structure that maps well to a given target architecture. An important example is parallelization: Spiral rewrites formulas to obtain the right form and the right degree of parallelism.

Spiral's rewriting system for parallelization [20,23] consists of three components to accomplish this goal:

- *Tags* encode target architecture types and parameters. Specifically, Spiral uses the tags "vec($\nu$)" for SIMD vector extensions and "smp($p, \mu$)" for shared memory.
- *Base cases* encode formula constructs that can be mapped well to a given target architecture. For instance, we denote a $p$-way parallel base case generalizing (6) with $\mathrm{I}_p \otimes_\parallel A_n$, using the tagged operator "$\otimes_\parallel$;" $A_n$ is any $n \times n$ matrix expression.
- *Rewriting rules* encode how to translate general formulas into base cases. For instance, the rewriting rule

$$\underbrace{\mathrm{I}_m \otimes A_n}_{\mathrm{smp}(p,\mu)} \rightarrow \mathrm{I}_p \otimes_\parallel (\mathrm{I}_{m/p} \otimes A_n) \tag{9}$$

  applies identity (8), but "knows" (due to the tag smp($p, \mu$)) that the target architecture is a $p$-way parallel shared memory system. It thus picks the parameter in (8) properly and steers the rewriting to the parallel base case.

**Vector SIMD instructions.** To generate efficient SIMD vector code (for instance, for Intel's SSE and AVX, AMD's AltiVec, IBM's VMX) we need to guarantee that all memory accesses are properly aligned and load/store whole vectors. All arithmetic should be done using vector operations and all data shuffling should take place in vector registers using efficient shuffle instructions (which may differ across supported SIMD architectures). The number of shuffle operations should be minimized. We introduce the tag "vec($\nu$)" for vector SIMD operation using $\nu$-way vector instructions. The construct

$$A \vec{\otimes} \mathrm{I}_\nu \tag{10}$$

(using the tagged operator $\vec{\otimes}$) can be implemented solely using vector arithmetic and guarantees aligned memory access of whole vectors. Further, Spiral automatically builds a library of vectorized implementations of permutations for every supported vector architecture. [23]The construct (10) and the vectorized permutations constitute some of the base cases for vector architectures. Vector rewriting rules like

$$\underbrace{A \otimes \mathrm{I}_n}_{\mathrm{vec}(\nu)} \rightarrow (A \otimes \mathrm{I}_{n/\nu}) \vec{\otimes} \mathrm{I}_\nu \quad \text{and} \quad \underbrace{\mathrm{I}_m \otimes A}_{\mathrm{vec}(\nu)} \rightarrow \underbrace{\mathrm{L}^{mn}_n}_{\mathrm{vec}(\nu)} \left( (A \otimes \mathrm{I}_{n/\nu}) \vec{\otimes} \mathrm{I}_\nu \right) \underbrace{\mathrm{L}^{mn}_m}_{\mathrm{vec}(\nu)} \tag{11}$$

are parameterized by $\nu$ and encode how to turn general constructs into vector base cases; $\mathrm{L}^{mn}_n$ and $\mathrm{L}^{mn}_m$ need more rewriting. Using vector base cases and vector breakdown rules, Spiral successfully vectorizes a large class of linear transform algorithms. Further details on Spiral's vectorization process can be found in. [18]

**Multicore and SMP.** Parallel program in our context can be implemented using data-parallel language extensions like OpenMP or a threading library like pthreads. Beyond extracting parallelism, the goal on symmetric multiprocessors and multicore CPUs is to balance the computational load and to avoid false sharing (private data of multiple processors stored in the same cache line). We aim at generating programs in which each cache line is accessed only by one processor at a time (the processor "owns" that cache line), and change of ownership happens as little as possible, thus minimizing communication invoked by the cache coherency protocol. We introduce the tag "smp($p, \mu$)" for shared memory computation on $p$ processors with cache line size $\mu$. The constructs

$$\mathrm{I}_p \otimes_\parallel A \quad \text{and} \quad P \bar{\otimes} \mathrm{I}_\mu, \quad P \text{ a permutation}, \tag{12}$$

expresses embarrassingly parallel, load balanced operation on $p$ processors and ownership change of cache lines, respectively; both can easily be translated into shared memory code. The shared memory rule set consists of rules like

$$\underbrace{A_m \otimes \mathrm{I}_n}_{\mathrm{smp}(p,\mu)} \rightarrow \underbrace{\left(\mathrm{L}^{mp}_m \otimes \mathrm{I}_{n/p}\right)}_{\mathrm{smp}(p,\mu)} \left(\mathrm{I}_p \otimes_{\|} (A_m \otimes \mathrm{I}_{n/p})\right) \underbrace{\left(\mathrm{L}^{mp}_p \otimes \mathrm{I}_{n/p}\right)}_{\mathrm{smp}(p,\mu)} \tag{13}$$

and allows to parallelize even small transforms and to produce efficient multicore code. Further information on Spiral's shared memory and multicore code generation can be found in. [24]

## 4. AUTOMATIC GENERATION OF SAR ALGORITHMS

In this section we describe how we formalize the PFA SAR algorithm described in Section 2. We first express the algorithm as breakdown rules, and subsequently discuss the formal parallelization, vectorization, and memory hierarchy optimization that Spiral applies automatically.

### 4.1 SAR Breakdown rules

The Polar formatting SAR[1] with the parameters chosen in our implementation is an linear operator and thus can be represented as matrix-vector product. More general interpolators would require the generalized approach Spiral based on the operator language.[20]

**Operator definition.** The SAR geometry and interpolators described in Section 2 are formally described by two high-level parameters: a *scenario* (geometry) $s = (m_1, n_1, m_2, n_2)$, and an *interpolator* $\alpha = (k, r, m, n, \alpha_r, \alpha_a)$, which parameterize the SAR operator $\mathrm{SAR}_{s,\alpha} \in \mathbb{C}^{m_1 n_1 \times m_2 n_2}$. The matrix-vector product

$$y = \mathrm{SAR}_{s,\alpha}\, x$$

performs the polar formatting operation on an linearized input matrix $x \in \mathbb{C}^{\,m_1 \times n_1}$ and produces a linearized output matrix $y \in \mathbb{C}^{m_2 \times n_2}$. The geometry $s$ describes the input and output geometry. $k$, $r$, $m$ and $n$ are the segment parameters: One segment interpolates from $m$ into $n$ points with $k$-fold upsampling and $r$ points overlap. We perform segmented nearest neighbor interpolation with upsampling, and $\alpha_r$ and $\alpha_a$ are functions that allow to compute base and stride as function of range or cross-range line number and segment number for the nearest neighbor interpolation, based on the scenario and geometry.

**PFA SAR breakdown rules.** With these definitions, we can express the polar format SAR algorithm by the following breakdown rules:

$$\mathrm{SAR}_{s,\alpha} \rightarrow \mathrm{DFT}_{m_2 \times n_2}\, \text{2D-Intp}^{m_1 \times n_1 \rightarrow m_2 \times n_2}_{(k,r,m,n,\alpha_r,\alpha_a)} \tag{14}$$

$$\text{2D-Intp}^{m_1 \times n_1 \rightarrow m_2 \times n_2}_{(k,r,m,n,\alpha_r,\alpha_a)} \rightarrow \left(\mathrm{Intp}^{n_1 \rightarrow n_2}_{(k,r,m,n,\alpha_a(i))} \otimes_i \mathrm{I}_{m_2}\right) \left(\mathrm{I}_{n_1} \otimes_i \mathrm{Intp}^{m_1 \rightarrow m_2}_{(k,r,m,n,\alpha_r(i))}\right) \tag{15}$$

$$\mathrm{Intp}^{u \rightarrow v}_{(k,r,m,n,w)} \rightarrow \left(\mathrm{I}_\ell \otimes_j \mathrm{Intp}^{m \rightarrow n}_{(k,w(j))}\right) \left(\mathrm{I}_\ell \otimes^r \mathrm{I}_m\right) \tag{16}$$

$$\mathrm{Intp}^{m \rightarrow n}_{(k,(b,s))} \rightarrow G^{km \rightarrow n}_{(b,s)}\, \mathrm{iDFT}_{km}\, \mathrm{S}_{\langle 0,k-1\rangle \otimes (m/2)}\, \mathrm{DFT}_m \tag{17}$$

$$\mathrm{iDFT}_{km}\, \mathrm{S}_{\langle 0,k-1\rangle \otimes (m/2)} \rightarrow \mathrm{iPDFT}^{\langle 0,k-1\rangle \otimes (m/2)}_{km} \tag{18}$$

We now describe the above rules and their meaning from the SAR algorithm perspective. (14) states that we implement polar formatting SAR, i.e., first perform a 2D interpolation and then a 2D DFT.

(15) states that we are using a separable 2D interpolator, and first interpolate in the range direction, followed by cross-range interpolation. The symbol $\otimes_i$ is a generalized tensor product notation that defines the index $i$ to be used in the expression.[25] It allows to have a index-dependent kernel in the tensor product while still capturing the tensor structure. In (15) the index $i$ is used to partially evaluate $\alpha_a$ and $\alpha_r$.

(16) states that we break lines into $\ell$ segments with overlap of $r$ for interpolation. The overlapped tensor product [25] $\otimes^r$ captures the overlap; it duplicates the $r$ overlapped elements (producing $\ell$ separate segments of length $m$) before sending them into the respective interpolation kernels. The evaluation of $w(j)$ produces a tuple of real numbers, $(b, s)$, which are used as base and stride for the nearest neighbor interpolation. We omitted the additional complications due to zero padding required in the last segment if the segment length does not divide the line length (taking overlap into account).

(17) states that a segment of length $m$ gets interpolated into $n$ points using $k$-fold upsampling (zero-padding in the frequency domain. The gather operation $G_{(b,s)}^{m\to n}$ performs the nearest-neighbor interpolation. It is defined as

$$G_{(b,s)}^{m\to n} = \left[ e^n_{\lfloor b+1/2 \rfloor} | e^n_{\lfloor b+s+1/2 \rfloor} | e^n_{\lfloor b+2s+1/2 \rfloor} | \dots | e^n_{\lfloor b+(n-1)s+1/2 \rfloor} \right], \quad b, s \in \mathbb{R}^+. \tag{19}$$

(18) states that the zero padding in the frequency domain is achieved using a pruned DFT, avoiding unnecessary computation and data initialization. The segment length $m/2$ allows for algorithmic freedom in the Cooley-Tukey pruned FFT algorithm (4).

Together with (1), (2), and (4), rules (14)–(18) are a formal specification of the PFA SAR algorithm.

### 4.2 Formal optimization

In this section we explain how an optimized formula for the SAR algorithm is constructed by Spiral. The input to our formal optimization is the problem specification,

$$\underbrace{\mathrm{SAR}_{s,\alpha}}_{\mathrm{smp}(p,\mu),\, \mathrm{vec}(\nu)} \quad . \tag{20}$$

It states that a polar formatting SAR formula for parameters $s$ and $\alpha$ should be found, subject to parallelization for $p$ processors and cache line length $\mu$, and vectorization for a $\nu$-way SIMD vector extension. We now discuss the formal optimization for memory hierarchy, parallelization, and vectorization.

**Memory hierarchy.** (14) introduces two main passes through the data (one for the 2D DFT and one for interpolation). However, since both the 2D DFT and the interpolation are separable, they require two passes each, as described by (2) and (15). In effect, both passes first process the rows and subsequently the columns of the data matrix. Applying the following formula manipulation rules (21)–(24) reduces the data movement to three round trips by merging the middle two passes.

$$A^\top \quad \to \quad A \quad \text{for symmetric } A \tag{21}$$
$$(AB)^\top \quad \to \quad B^\top A^\top \tag{22}$$
$$(A \otimes B)^\top \quad \to \quad A^\top \otimes B^\top \tag{23}$$
$$(A \otimes B)(A \otimes_i C) \quad \to \quad A \otimes_i (BC) \quad \text{for compatible } B \text{ and } C \tag{24}$$

(21)–(23) applied to (2) derives the *column row* 2D DFT algorithm, and (24) merges the column (cross-range) interpolation with the first (column) stage of the 2D DFT,

$$(\mathrm{DFT}_{n_2} \otimes \mathrm{I}_{m_2}) \left( \mathrm{Intp}^{n_1\to n_2}_{(k,r,m,n,\alpha_a(i))} \otimes_i \mathrm{I}_{m_2} \right) \to \left( \left( \mathrm{DFT}_{n_2}\, \mathrm{Intp}^{n_1\to n_2}_{(k,r,m,n,\alpha_a(i))} \right) \otimes_i \mathrm{I}_{m_2} \right).$$

Strided memory access can cause severe performance degradation due to cache thrashing. Our formulas contain constructs $A \otimes \mathrm{I}_n$ for $A \in \mathbb{C}^{k\times m}$ which are candidates for this problem. However, using the formula identity

$$A \otimes \mathrm{I}_n = (A \otimes \mathrm{I}_{n/\mu}) \otimes \mathrm{I}_\mu$$

we can prove that as long as the last cache level can hold a $m$ cachelines at stride $n/\mu$ (for the input vector $x$, in stride cache lines) plus $k$ cache lines at stride $n/\mu$ (for the output vector $y$, in stride cache lines), no cache line that is evicted will be reloaded in the same stage. Thus, all data makes only one round trip memory-cache-memory in the computation $y = (A \otimes \mathrm{I}_n)x$. For $\mathrm{I}_n \otimes A$ only a cache capacity of $m + k$ complex numbers is required since data is stored contiguously in memory and no cache effects are visible across iteration boundaries. In Section 6 we show that for our target machines and scenarios these conditions hold true, and thus straight-forward loop code will deliver full performance; no special optimizations like buffering or blocked corner turns are required.

**Shared memory parallelization.** The coarse structure of any formula for (20) derived with our rule system only contains top level constructs

$$\underbrace{A_i \otimes_i \mathrm{I}_n}_{\mathrm{smp}(p,\mu)} \quad \text{and} \quad \underbrace{\mathrm{I}_n \otimes_i A_i}_{\mathrm{smp}(p,\mu)}, \tag{25}$$

with all the details (and vectoization tags) hidden in the respective $A_i$. It is thus sufficient to parallelize (25). The system follows the approach discussed in,[16] which was overviewed in Section 3.2. In particular, (9) and (13) lead to data parallel

loops which only require a global barrier at its end. The data is distributed across processors so that all processors always works on contiguous data; we fully avoid false sharing.

**Vectorization.** Similar to the previous section on parallelization, it is sufficient to discuss the vectorization of

$$\underbrace{A_i \otimes_i \mathrm{I}_n}_{\mathrm{vec}(\nu)} \quad \text{and} \quad \underbrace{\mathrm{I}_n \otimes_i A_i}_{\mathrm{vec}(\nu)}. \tag{26}$$

Using the propagation rule

$$\underbrace{\mathrm{I}_n \otimes_i A_i}_{\mathrm{vec}(\nu)} \rightarrow \mathrm{I}_n \otimes_i \underbrace{A_i}_{\mathrm{vec}(\nu)}$$

finally—after applying all breakdown rules (1)–(24)—requires to formally vectorize

$$\underbrace{\mathrm{DFT}_n}_{\mathrm{vec}(\nu)} \quad \text{and} \quad \underbrace{\mathrm{iPDFT}_n^\sigma}_{\mathrm{vec}(\nu)}.$$

The problem to handle is to vectorize a single DFT or pruned DFT with data contiguous in memory, which is done using the techniques discussed in.[18,20] This approach vectorizes computation along range lines. For the cross-range computation the rewriting process arrives at

$$\underbrace{\mathrm{DFT}_n \otimes \mathrm{I}_\nu}_{\mathrm{vec}(\nu)} \rightarrow \mathrm{DFT}_n \,\vec{\otimes}\, \mathrm{I}_\nu \quad \text{and} \quad \underbrace{\mathrm{iPDFT}_n^\sigma \otimes \mathrm{I}_\nu}_{\mathrm{vec}(\nu)} \rightarrow \mathrm{iPDFT}_n^\sigma \,\vec{\otimes}\, \mathrm{I}_\nu$$

which applies (11). This approach packs the data of $\nu$ DFTs into vectors, and performs a "vector DFT." The remaining construct to handle is the nearest neighbor gather operation,

$$\underbrace{G_{b_i,s_i} \otimes_i \mathrm{I}_\nu}_{\mathrm{vec}(\nu)} \rightarrow G_{b_i,s_i} \,\vec{\otimes}_i\, \mathrm{I}_\nu,$$

which is handled as new vector base case. The implementation performs $\nu$ gathers, each with its own $b_i$ and $s_i$. Picking appropriate $b_i$ and $s_i$, this construct can implement all gathering needed by the range and cross-range nearest neighbor interpolation.

## 5. VIRTUAL MEMORY OPTIMIZATIONS

SAR's large input data sizes (over 3.4 GB in the case of the 100 Megapixel scenario) strain the entire memory hierarchy. While we can optimize for certain features of the memory hierarchy (notably L1/L2 cache) through Spiral's rewriting and exploration mechanism, other features must be optimized through a systems approach. We detail such systems optimizations here.

**Large Pages.** While data and instruction caches are well known features of the memory hierarchy, caches that store virtual-to-physical address translations such as Translation Lookaside Buffers (TLBs) and Paging-Structure caches tend to be more arcane. These caches are often organized into their own hierarchies and their performance implications are typically harder to quantify and subject to change between processor revisions. Nevertheless, as we show below, their impact on performance for large data sets is considerable.

The default x86 TLB entry stores a translation for a 4 KB range of virtual addresses i.e. a range of consecutive 48-bit virtual addresses which share the same 36-bit prefix. TLBs tend to be small, on the order of a few hundred entries. This equates to a virtual address space range of 1–2 MB. Caches of this size incur frequent misses when serving a 3.4 GB data set. Unlike data or instruction cache misses, TLB misses tend to stall the CPU while it is forced to undertake a long latency walk through a hierarchy of Page Tables. Though the expense of Page Table walks can be mitigated by Paging-Structure caches, TLB misses remain costly for large data sets. Fortunately, the x86 exposes so-called Large Pages which contain translations for a 2 MB range of virtual addresses, i.e., a range of addresses which share the same 27-bit prefix. Though the TLB contains fewer Large Page entries, the address coverage is on the order of 128 MB or greater. Large Pages not only reduce the number of TLB misses and thus Page Table walks but also the latency of Page Table walks as fewer levels of the Page Table hierarchy need to be traversed.

**Operating System.** An application wishing to take advantage of Large Pages requires support from the Operating System and middleware. Linux kernels since at least 2.6.16 offer support for Large Pages through the `hugetlbfs` pseudo-filesystem. Large Pages are best allocated at boot time by the kernel as they require large contiguous regions and must co-exist with 4 KB pages. We use IBM's contributed `libhugetlbfs` which offers plug-in replacements for C Standard Library memory allocation functions `{m,c,re}alloc` that use Large Pages as backing store for the heap. There is additional support for mapping the executable's text and data segments to Large Pages through a special linker script. Despite an executable size exceeding 6 MB for a 100 Megapixel scenario, we found no appreciable difference in runtimes when text and data segments were mapped to Large Pages.

## 6. EXPERIMENTS

**Scenarios.** We examined two PFA SAR scenarios. Both scenarios were derived from [6] for comparison purposes. The first scenario is 4k x 4k (16 Megapixel) in size and the second scenario is 10k x 10k (100 Megapixel). Both scenarios have longer slant ranges (24 km and 200 km respectively), fine resolution (.1 m and .3 m) and small coherent integration angles (approx. $4\,^\circ$ and $7\,^\circ$). In both cases, the input size is slightly larger than the output size on account of re-gridding. The data sets for both scenarios consist of single precision floating point complex numbers stored in an interleaved format.

**Methodology.** Spiral was installed on and generated code for three generations of 64-bit Intel Quad Core CPUs: the 3.0 GHz 5160, the 3.0 GHz X9560, and the 2.66 GHz Core i7 920. The first two processors share the same microarchitecture (Core 2 Quad) but were fabricated using different process technologies (65 nm and 45 nm). The X9560 also boasts a faster vector shuffle unit. Intel's first "native" Quad Core, the Core i7, is mainly distinguished by having 2–3 times the aggregate (read and write) main memory bandwidth of the Core 2 due to a triple channel, on-die memory controller. The vital L2 data cache size figure varies across our range of CPUs with capacities of 4 MB, 6 MB, and 8 MB respectively.

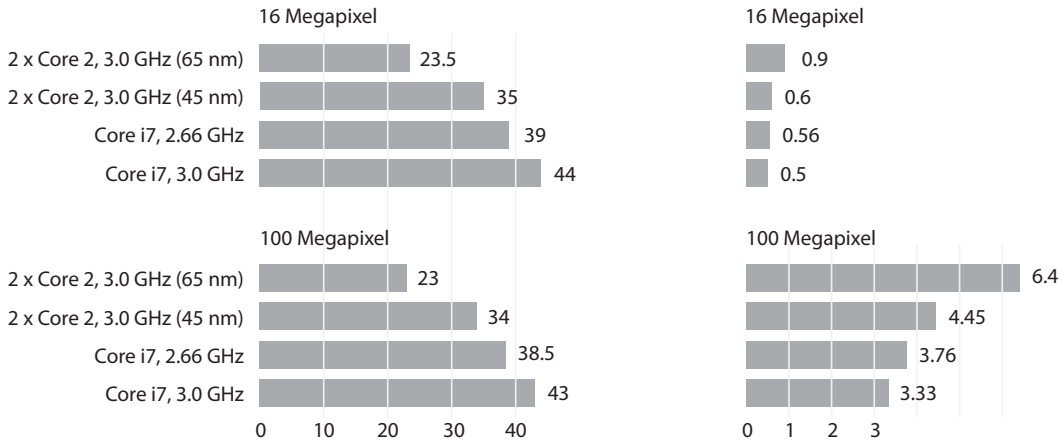
Spiral conducted an exhaustive search for each CPU and Intel's `icc` C compiler (version 11.0.074), invoked using the `-O3` option, was used for the generated code. The generated code has a standard library function interface which takes an input array and an output array. 2-way complex SSE vector instructions were leveraged using the SSE intrinsics exposed by `icc`. We exploited multicore through the POSIX pthreads API which we combined with a custom barrier instruction. We had hoped to generate OpenMP versions of our code but the large basic blocks in the generated code (a consequence of Spiral's essential loop unrolling optimization) exceeded the code size limits of Intel's OpenMP compiler.

Code timing was conducted by sandwiching a sequence of repeated function invocations between a pair of x86 `rdtsc` instructions. `rdtsc` returns the number of CPU ticks (cycles) since reset. The timing sequence is preceded by an Intel recommended "warmup" instruction sequence. The average elapsed time for a function invocation is determined by dividing the number of ticks accumulated between the `rdtsc` instructions by the number of function invocations. All code was generated and executed under Linux 2.6.18+ with 8 GB of RAM for the 5160 and X9560 and 12 GB of RAM for the Core i7. We used large, 2 MB, pages in the main experiment. The performance gain compared to standard 4 KB pages is also shown below.

We report performance figures in pseudo-Gflop/s (a typical metric for transforms) using an instruction count derived from the computational cost analysis in Section 2. For grid computation, we experimented with two methods: precomputation and on-the-fly computation. For the transcendental functions `sin, cos` we used Intel's Vector Math Library. As grid computation is linear in the dimensions of the scenario, we observed no difference in performance between the two methods.

**Results and Analysis.** Figure 5(a) shows the obtained performance and Figure 5(b) the corresponding runtime that we achieved with Spiral-generated code. For clockspeed equivalency, we also show projected performance on a 3.0 GHz Core i7 assuming a speedup that scales linearly with CPU frequency. The performance leap between the 5160 and X9560 is mainly due to the improved vector shuffle unit that Spiral could target with its vector backend and the larger L2 capacity which enhances the benefit of our data layout strategies by enabling us to hold more range/cross-range lines in cache. This trend continues on the Core i7 with its even larger L2. Performance is further improved by the increased memory bandwidth which aids the memory bound 2D FFT.

Runtimes for the more recent X9560 are competitive with those presented in [6] despite the Cell's considerable memory bandwidth advantage (25.6 GB/sec vs. 6.4 GB/sec). The Core i7's improved memory bandwidth enables a 100 Megapixel runtime that edges out the Cell's by .5 seconds. As we alluded to, the performance improvements attained with Large Pages were pronounced as shown in Figure 6. The dramatic speedup with Large Pages on the 5160 relative to the X9560 and Core i7 was determined (through analysis of Intel CPU Performance Counters using Intel's VTune) to be caused by a Page Walk latency that was on average 2.6 times (248 cycles) longer on the 5160 than on the X9560 (95 cycles). The




(a) Performance in Gflop/s (b) Runtime in seconds

Figure 5. Performance and corresponding runtime of computer-generated SAR implementations on various Intel quadcore systems with large pages. The results on Core i7, 3.0 GHz are "virtual," i.e., obtained by linearly scaling the measured results on Core i7, 2.66 GHz.

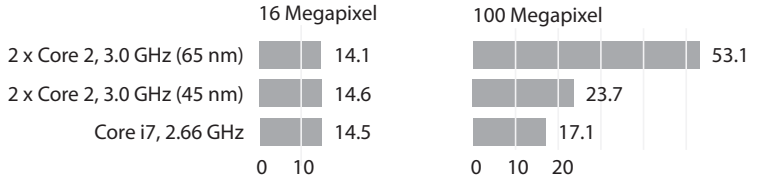



Figure 6. Performance (Gflop/s) improvement in percent by using 2 MB pages versus 4 KB pages (the default).

reduction of Page Walks and total Page Walk latency through the use of Large Pages produces the 50% speedup on the 5160 due to the relative slowness of its 4 KB Page Walks.

Finally, we note that the code generation time was about 5–10 hours for each combination of scenario and platform.

## 7. CONCLUSIONS

We believe to have made two main contributions. First, we have prototypically shown that it is possible to replace the human programmer in developing the computational core of high-performance SAR implementations on state-of-the-art multicore platforms. The key to this success is an extension of the Spiral framework that we have shown. Second, our results show that commodity architectures, if properly used, have become a viable option for performance demanding embedded signal processing functions. Future architectures seem poised to continue this trend with ever larger vector widths (Intel's 8-way AVX has been announced), and increasing number of cores. Forthcoming General Purpose Graphics Processing Units take these features to new extremes. Consequently, automatic code generation becomes urgently more relevant; we hope to demonstrate Spiral's effectiveness on these emerging architectures using PFA SAR Image Formation as one benchmark.

## REFERENCES

[1] Carrara, W. G., Goodman, R. S., and Majewski, R. M., [*Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*], Artech House (1995).

[2] Thompson, P., Wahl, D. E., Eichel, P. H., Ghiglia, D. C., and Jakowatz, C. V., [*Spotlight-Mode Synthetic Aperture Radar: A Signal Processing Approach*], Kluwer Academic Publishers, Norwell, MA, USA (1996).

[3] Martin, G. D. and Doerry, A. W., "Sar polar format implementation with matlab," Tech. Rep. SAND2005-7413, Sandia National Laboratories (2005).

[4] Franchetti, F., de Mesmay, F., McFarlin, D., and Püschel, M., "Operator language: A program generation framework for fast kernels," in [*IFIP Working Conference on Domain Specific Languages (DSL WC)*], (2009).

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

[6] Rudin, J., "Implementation of polar format SAR image formation on the IBM cell broadband engine," in [*High Performance Embedded Computing (HPEC)*], (2007).

[7] West, J. M., Li, H., Vanichayobon, S., Muehring, J. T., Antonio, J. K., and Dhall, S. K., "A hybrid fpga/dsp/gpp prototype architecture for sar and stap," in [*High Performance Embedded Computing (HPEC)*], (2000).

[8] Bueno, D., Conger, C., George, A. D., Troxel, I., and Leko, A., "Rapidio for radar processing in advanced space systems," *Trans. on Embedded Computing Sys.* (2008).

[9] Conti, A., Cordes, B., Lesser, M., and Miller, E., "Backprojection and synthetic aperture radar processing on a hhpc," in [*Research and Industrial Collaboration Conference: Center For Subsurface Sensing and Imaging Systems*], (2005).

[10] Kepner, J., Currie, T., Kim, H., Matthew, B., Mccabe, A., Moore, M., Rabinkin, D., Reuther, A., Rhoades, A., Tella, L., , and Travinn, N., "Deployment of SAR and GMTI signal processing on a boeing 707 aircraft using pmatlab and a bladed linux cluster," in [*High Performance Embedded Computing (HPEC)*], (2004).

[11] McCoy, D., Bergmann, J., and Seefeld, S., "Sourcery vsipl++ on the cell broadband engine: A fused fast convolution example," in [*HPCMP-UGC '07: Proceedings of the 2007 DoD High Performance Computing Modernization Program Users Group Conference*], (2007).

[12] Backues, M., Majumder, U., York, D., and Minardi, M., "Synthetic aperture radar backprojection on sony playstation 3 cell broadband engine and intel quad-core xeon," in [*High Performance Embedded Computing (HPEC)*], (2008).

[13] Dershowitz, N. and Plaisted, D. A., "Rewriting," in [*Handbook of Automated Reasoning*], Robinson, A. and Voronkov, A., eds., **1**, ch. 9, 535–610, Elsevier (2001).

[14] Franchetti, F., Kral, S., Lorenz, J., and Ueberhuber, C., "Efficient utilization of SIMD extensions," *Proceedings of the IEEE* **93**(2) (2005). special issue on "Program Generation, Optimization, and Adaptation".

[15] Xiong, J., Johnson, J., Johnson, R., and Padua, D., "SPL: A language and compiler for DSP algorithms," in [*Proc. PLDI*], 298–308 (2001).

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

[17] Franchetti, F. and Püschel, M., "A SIMD vectorizing compiler for digital signal processing algorithms," in [*Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*], 20–26 (2002).

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

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

[20] Franchetti, F. and Püschel, M., "Generating high-performance pruned FFT implementations," in [*International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*], (2009).

[21] Franchetti, F., Voronenko, Y., Milder, P. A., Chellappa, S., Telgarsky, M., Shen, H., D'Alberto, P., de Mesmay, F., Hoe, J. C., Moura, J. M. F., and Püschel, M., "Domain-specific library generation for parallel software and hardware platforms," in [*NSF Next Generation Software Program Workshop (NSFNGS) colocated with IPDPS*], (2008).

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

[23] Franchetti, F. and Püschel, M., "Generating SIMD vectorized permutations," in [*International Conference on Compiler Construction (CC)*], *Lecture Notes in Computer Science* **4959**, 116–131, Springer (2008).

[24] Franchetti, F., Voronenko, Y., and Püschel, M., "A rewriting system for the vectorization of signal transforms," in [*Proc. High Performance Computing for Computational Science (VECPAR)*], (2006).

[25] Gacic, A., *Automatic Implementation and Platform Adaptation of Discrete Filtering and Wavelet Algorithms*, PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University (2004).